2020

High Performance and Secure Execution Environments for Emerging Architectures

Mazen Alwadi
University of Central Florida

HIGH PERFORMANCE AND SECURE EXECUTION ENVIRONMENTS FOR EMERGING ARCHITECTURES

by

MAZEN ALWADI
M.Sc. Yarmouk University, 2017
B.Sc. Jordan University of Science and Technology, 2012

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2020

Major Professor: Amro Awad

ABSTRACT

Energy-efficiency and performance have been the driving forces of system architectures and designers in the last century. Given the diversity of workloads and the significant performance and power improvements when running workloads on customized processing elements, system vendors are drifting towards new system architectures (e.g., FAM or HMM). Such architectures are being developed with the purpose of improving the system's performance, allow easier data sharing, and reduce the overall power consumption. Additionally, current computing systems suffer from a very wide attack surface, mainly due to the fact that such systems comprise of tens to hundreds of sub-systems that could be manufactured by different vendors. Vulnerabilities, backdoors, and potentially hardware trojans injected anywhere in the system form a serious risk for confidentiality and integrity of data in computing systems. Thus, adding security features is becoming an essential requirement in modern systems.

In the purpose of achieving these performance improvements and power consumption reduction, the emerging NVMs stand as a very appealing option to be the main memory building block or a part of it. However, integrating the NVMs in the memory system can lead to several challenges. First, if the NVM is used as the sole memory, incorporating security measures can exacerbate the NVMs' write endurance and reduce its lifetime. Second, integrating the NVM as a part of the main memory as in DRAM-NVM hybrid memory systems can lead to higher performance overheads of persistent applications. Third, Integrating the NVM as a memory extension as in fabric-attached memory architecture can cause a high contention over the security metadata cache. Additionally, in FAM architectures, the memory sharing can lead to security metadata coherence problems. In this dissertation, we study these problems and propose novel solutions to enable secure and efficient integration of NVMs in the emerging architectures.

This dissertation is dedicated to my parents, siblings, and advisor the people who supported me through this journey and made it possible to achieve my goal.

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Dr. Amro Awad, for his guidance and directions on my research, his instructions on my writing and presentation skills, and his tremendous efforts on reviewing and commenting on my papers. Without his help, I do not think that I could have achieved this. From the external collaborators, I would like to thank Dr. Rujia Wang from Illinois Institute of Technology, Dr. Clayton Hughes and Dr. Simon Hammond from Sandia National Laboratories, Dr. John Seel from the Navy. Furthermore, I would like to express my gratitude to the committee members, Dr. Wei Zhang, Dr. David Mohaisen for his support, help, and direction on multiple research papers. Dr. Ricard Ewetz, for his directions and comments during the meetings we had in Dr. Awad's office. Dr. Fan Yao, for his insightful discussions and suggestions during the reading group seminars, and during the IISWC conference. Additionally, I would like to express my appreciation to my labmates: Kazi Zubair, Vamsee Reddy, Mao Ye, John McFarland, and Abdullah Al-Arafat. We shared wonderful times that I would cherish as long as I live. Furthermore, I would like to thank my friends here in Orlando, who stood by my side and helped me during this journey, namely, Mustafa Rawashdeh, Anas Mstareehi, and Mohammed Obeidat. Finally, the best for the last as we say, I would like to express my gratitude and appreciation to my parents and siblings. The ones who never let me down, stood by my side at all times, and provided all kinds of support through my whole life journey. I would like to seize this opportunity to thank you from the bottom of my heart, and say you are the joy of my life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Non-Volatile Memories (NVMs) are emerging as promising contenders for DRAM, and promise to provide terabytes of persistent data capacity that can be accessed using regular load and store operations [50,53]. Secure NVM systems commonly aim to protect confidentiality, integrity, and availability of the data. While data persistency is an attractive feature that enables persistent applications, e.g., filesystems and checkpointing, it also facilitates data remanence attacks [18,24,90,93]. To protect the NVM's data at rest, encryption becomes a necessity. Encryption targets the confidentiality among the security requirements. Additionally, the NVMs' data persistency comes at the cost of 3-4x slowdown, due to the higher access latencies of the NVM. In this chapter, we briefly discuss the crash consistency problem when NVMs are integrated as a sole main memory in Section 1. In Section 1, we discuss the performance challenges of persistent applications in hybrid-memory systems. Finally, in Section 1, we discuss the challenges of implementing secure-memory measures in fabric-attached memories.

Improving the Performance and Reducing the Writes for Tree of Counters Integrity Protected Systems

Due to the nature of the Trees of Counters (ToCs), they cannot be recovered from leaves. Thus, and unlike general Merkle Trees (MTs), it is insufficient to just rely on persisting the leaves and rebuilding the tree after a crash [99]. Meanwhile, strictly persisting tree updates for each memory write can incur a significant write overhead, hence reducing the memory lifetime and significantly degrading the performance. Finally, rebuilding the tree after a crash can take hours, if we are unable to identify the subset of tree nodes that have been possibly lost. Recent work [99], demonstrated that a practical NVM size (e.g., 8TB) would require 7.8 hours for recovery. On the other hand,

high-availability systems have stringent requirements of 99.999% (five nines rule), i.e., the system can sustain a total of 7.8 hours down time only once each 89 years.

The state-of-the-art scheme Anubis [99], addresses the recovery time problem by persistently tracking the addresses of security metadata currently in the cache and thus limiting recovery of metadata to these addresses only. By doing so, Anubis no longer needs to rebuild the whole tree but only a subset of nodes that have been potentially updated and lost due to a crash. To enable ToC's recovery, Anubis uses a lazy update scheme while persisting each cache update to the NVM. Thus, Anubis only needs to verify that the shadowed cache (in NVM) has not been tampered with after a crash and then load that shadow cache content into the cache. By doing so, Anubis is able to recover the ToC but at the cost of a write to the shadow region on each security metadata cache update. Although the first solution capable of recovering ToC integrity trees, the overhead of Anubis can limit its deployment.

NVMs' limited write endurance is perhaps the most challenging part towards its wide adoption [18,20,21,55,90,91,93]. In fact, encryption significantly exacerbates the write endurance issue due to the encryption's diffusion property [93]. Meanwhile, the state-of-the-art solution, Anubis [99], incurs 87% write overhead when used with ToC integrity trees, and systems using Anubis are expected to have almost half the lifetime span of systems without Anubis but no recoverability. Moreover, NVM writes are power consuming and have much higher latency than the reads [24,49, 93]. Obviously, doubling the write's bandwidth, as incurred by Anubis, limits its deployment and hence leaves NVMs in unrecoverable state.

In Chapter 2, we aim to bridge the gap between recoverability and high-performance for secure NVM systems. We mainly focus on ToC integrity trees, due to its commercial adoption (Intel processors) and security advantages, including resistance to tampering and replay attacks. To bridge the aforementioned gap, we propose Phoenix, a novel memory controller design that achieves both

recoverability and high-performance of secure NVM systems. Phoenix is based on our observation that reconstructing the exact content of the security metadata cache after a crash does not require shadowing all the cache updates, i.e., allowing imprecise content addresses shadow of the metadata cache could be sufficient. In fact, we can reconstruct the exact lost cache state after recovery by recalculating the potentially lost values and then verify the integrity of the reconstructed cache. By relying on the ability to recover the tree leaves, only a small subset of the cache updates need to be persisted in memory. Meanwhile, we still can verify that the recovered cache content reflects exactly the same cache state before the crash. Our optimization, realized in Phoenix+, relaxes persisting encryption counters on eviction, to only persist encryption counters on the N-th write, reducing Phoenix's overhead significantly.

Improving the Performance of Persistent Applications in HMM

Emerging NVMs can be integrated as storage devices (e.g., inside Solid-State Drives), such as Intel's Optane Drive [4] or as part of the system's memory hierarchy. For integrating NVMs into the memory hierarchy, there are several standards and options [6, 8, 10]. Most notably, Intel's DIMM-like NVM modules (called Optane DC [8]) can be integrated either as the main memory, or as a part of the main memory along with other memory options (e.g. DRAM and HBM). When used as a part of the main memory, it can be exposed as a separate physical memory address range extending the physical address range of DRAM, or the DRAM can be used as a hardware-managed cache of the physical range of the Optane DC [8]. The former is called *application direct mode*, which is similar to exposing different memory zones to the system in Non-Uniform Memory Architectures (NUMA), whereas the latter is called *memory mode*. Memory mode gives up on the persistence feature, as memory blocks could be updated in the volatile DRAM when applications flush their updates from internal caches. However, since DRAM caches a large number of the

NVM pages, it significantly improves the access latency, especially for frequently-used pages. On the other hand, application direct mode ensures persistence of pages mapped to the NVM address range, but incurs significant latencies as it relies on the capacity-limited internal processor caches (not the external DRAM). Therefore, the current integration options for Optane memory modules as (part of) the main memory ignores the performance of persistent applications that require both persistence and high performance (e.g., cacheability in DRAM).

To enable fast persistent DRAM caching of NVM, we propose a novel memory controller design that leverages *selective NVM mirroring* for persistent pages cached in DRAM. Our design supports both memory mode and application direct mode, and transparently ensures durability of updates to persistent pages cached in DRAM. Moreover, our memory controller minimizes the number of writes to NVMs by relaxing the mirroring of DRAM cached pages' updates if their source pages in NVMs are in the logically non-persistent part of NVM (i.e., used for hosting pages that do not need to be persisted). Similar to memory mode support in current processors, our memory controller transparently migrates pages between NVM and DRAM. However, we ensure persistence of DRAM cached pages by inferring their semantic from their original address in NVM. Our scheme only incurs additional writes to DRAM if the page is cached there, in addition to the NVM write which would have occurred anyway. However, future reads will be served from DRAM, which enables fast and persistent caching of durable NVM pages. Additionally, by allowing persistent pages to be located in DRAM, our scheme leverages additional bank-level parallelism for accessing persistent pages, instead of forcing all accesses to NVM. Our scheme is similar in spirit to the write-through scheme typically used in internal processor caches, but involves novel optimizations and design considerations due to the nature of writes and how DRAM is exposed to the system (memory mode or application direct mode). While all prior work on persistent applications explored optimizations for writing to persistent objects, this is the *first work to explore optimizing the* ***read operations*** *of persistent objects*.

Secure Memory in FAM Architectures

Secure memory support is becoming an essential part of most modern processors [1, 7, 30]. For instance, AMD's Secure Memory Encryption (SME) and Intel's Software Guard Extension (SGX), are prominent examples of security measures that are being incorporated in modern processors. It is expected that the usage and importance of these security features will increase with time, due to the proliferation of edge devices and cloud computing [1]. However, the implementation of these features has been approached in a processor-centric way [15, 19–21, 77, 81, 90, 91, 97, 99, 101]. In particular, each processor chip handles the security operations (*e.g.*, encryption and integrity verification) for the memory modules directly attached to its channel(s), *i.e.*, the memory controller of each processor handles the security metadata and operations corresponding to the data residing on the modules directly attached to it. This design philosophy allowed independent implementations and processing of security metadata in multi-processor systems [72, 73]. As systems integrating fabric-attached devices become more common, we argue that these processor-centric implementations of secure memory must be revisited.

The challenges of implementing secure memory in FAM architectures are as follows. First, the memory pool is both shared and directly accessed by multiple PEs across multiple nodes. Thus, each PE must have the most recent version of the security metadata, in order to correctly decrypt the data and verify its integrity. While ensuring the freshness of the security metadata can be achieved using conventional coherence protocols, such protocols can significantly degrade the system's performance due to their incompatibility with the security metadata requirements. For instance, writing a cacheline back to the memory, requires writing its encryption counter and the full integrity tree branch alongside with the data [15, 19, 99]. Abiding by this correctness condition, requires notifying all the PEs about the update and waiting for acknowledgments before writing these updates to the memory. Second, to ensure the correctness of these updates and prevent data

races, the whole integrity tree branch should be locked until the whole update process is complete. Third, buffering these updates in the processor's write buffers until they are acknowledged, can introduce processor stalls due to filling these buffers. Additionally, as FAM standards do not mandate a specific memory technology, emerging Non-Volatile Memories (NVMs) are expected to be one of the most promising options to build FAM modules, due to persistence, ultra-low idle power, and massive capacities [50, 53]. Fourth, adding NVM to the picture exacerbates the updates' buffering problem, and introduces the atomicity and crash consistency requirements.

When taken together, sharing of FAM modules, the large number of security metadata updates, and NVM's atomicity requirements lead to an unorthodox security metadata coherence problem. For instance, using a conventional invalidation-based data coherence protocol renders the security metadata caches useless, due to the frequent invalidation of upper integrity tree nodes. On the other hand, an update-based data coherence protocol can lead to a significant increase in the system's traffic - each memory write can lead to tens of updates to security metadata [15, 19, 76, 77, 81]. Additionally, both schemes would have to atomically persist the updates to ensure the system's recoverability. We observe that using conventional coherence protocols can lead to an average of 57.1%, and up to 90.4% overhead. Unlike conventional data coherence, we first need to ensure that the updates are propagated and reflected into the integrity tree nodes in the same strict order that they were initiated. Such ordering is required to ensure the integrity tree reflects the correct memory state, which typically requires a (very hard to achieve) unified system clock. Second, the integrity and timeliness of the security metadata coherence messages need to be ensured, as tampering with/delaying a security metadata coherence message can lead to falsely declaring a legitimate update as an attack. Finally, we need to ensure the recoverability of each PE in the system, as if a single PE crashed before persisting its security metadata, it could lead to a whole system failure.

To address this problem, we propose a novel hardware support, Minerva, which enables both coher-

ent and crash-consistent views of security metadata in memory-centric architectures. In particular, Minerva exploits applications' data locality and integrity tree nodes' coverage to minimize the overhead of maintaining security metadata coherence. Furthermore, Minerva ensures security at the system level by enforcing exclusive caching of security metadata. In other words, Minerva only allows a single security metadata cacheline to be cached by one processing element at any point in time. To reduce the overhead of maintaining coherence, Minerva relies on a lazy-invalidate scheme to make the overhead a function of security metadata cache *misses* instead of security metadata cache *updates*. Finally, to reduce the communication overhead and facilitate scalability, Minerva implements an inexpensive directory-like scheme that leverages the unused bits in the parallelizable integrity tree to store the ID of the PE currently caching the node.

Dissertation Organization

The rest of the dissertation is organized as following. In Chapter 2, we discuss Phoenix, our memory controller design, which relaxes the number of ToC updates required to ensure the system's recoverability. In Chapter 3, we discuss Stealth-Persist, a memory controller design for hybrid-memory systems, which enables caching the persistent application's data in the DRAM while ensuring the persistency. Later, In Chapter 4, we discuss the Split-Tree scheme and the caching techniques that aims to reduce the overhead of implementing secure memory in fabric-attached memory architectures. Later, in Chapter 5, we discuss Minerva, a novel memory controller design that addresses the security metadata coherence problem in fabric-attached memories. Finally, in Chapter 6, we conclude the dissertation with a summary of the proposed schemes.

CHAPTER 2: IMPROVING THE PERFORMANCE AND REDUCING THE WRITES FOR TREE OF COUNTERS INTEGRITY PROTECTED SYSTEMS

Background and Motivation

In this Section, we review background and related concepts, then motivate our work. In particular, we start by defining the threat model, followed by all relevant concepts.

Threat Model

Similar to the state-of-the-art approaches [18, 34, 55, 71, 88, 90], our threat model considers the processor chip to be the secure boundary. The processor contains the root of the integrity tree and the encryption key, where everything outside the processor is considered untrusted. We assume an attacker is capable of performing passive and active attacks including bus snooping and replaying memory packets, can scan the memory contents, and may tamper with memory contents. We also assume the attacker can perform attacks while the system is either on or off. Access pattern leakage attacks, electromagnetic (EM) inference attacks, and differential power analysis attacks are, however, beyond the scope of this work.

Counter Mode Encryption

One of the major security vulnerabilities of NVM systems is the data remanence problem. Therefore, NVM is usually paired with encryption for data protection. The state-of-the-art secure processors (e.g., Intel Xeon Processor E-family) use counter-mode encryption, shown in Figure 2.1,

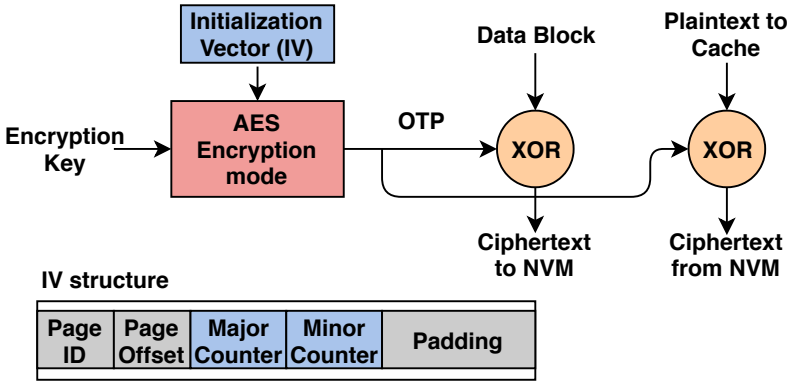

Figure 2.1: Counter-Mode Encryption in state-of-the-art secure memories.

since it provides strong defenses against a range of attacks (e.g., snooping, known plain-text, and dictionary-based). Moreover, the counter-mode has a smaller encryption/decryption overhead compared to other schemes due to overlapped latency of data fetching and one-time-pad generation [18, 21, 93]. For each write to a data block, its associated counter will be incremented by 1. The updated counter is used to generate an initialization vector that is used along with the processor key serve as inputs for the encryption engine to generate a One-Time-Pad (OTP). After being XOR'ed with this OTP, the data block is encrypted and can be saved in memory. Similarly, a read request uses the same encryption pad to generate plain-text for processors but without updating any counter value.

The size and organization of counters vary in different state-of-the-art schemes. The counters used in ToC are ***monolithic counters***, where each of 56-bit long associated with a data block and one 64B counter cacheline can accommodate counters for eight 64B memory blocks. Encryption counter overflow can be costly, and causes long system stalls which is generally unacceptable [71]. Therefore, the monolithic counter should be large enough to prevent overflowing, which means more storage overhead. For encryption/decryption, the monolithic counter will be padded with a block address to generate the initialization vector [71]. To encrypt/decrypt the data, secure

processors use AES counter-mode encryption. Figure 2.1 shows how counter code encryption works.

Several other state-of-the-art schemes use the ***split counter*** scheme [18, 19, 55, 76, 81, 90, 99], in which each data block is associated with one per-page major counter and one per-block minor counter. The major counter is shared by all the blocks within that page. Encryption/Decryption requires knowing both major and minor counter values to generate the OTP. Since each minor counter only accounts for seven bits, and the major counter for 64 bits, a small storage overhead occurs. However, when a minor counter overflows, the major counter is incremented by 1 and the whole page has to be encrypted using the new major counter [18, 19, 55, 76, 81, 90, 99].

Integrity Verification

Since the trusted boundaries are limited to the processor chip, whenever a block is fetched from the memory, the memory integrity needs to be verified. In state-of-the-art research and secure processors design [19, 71, 90, 99], the Merkle Tree – one of approaches used for ensuring integrity, is widely studied and used for memory integrity verification.

Basically, Merkle Tree is an N-ary hash tree where its leaves correspond to encryption counters for data blocks [71] and every N leaves will have a hash value calculated based on the counter values. Similarly, all the intermediate nodes up to the root are constructed using the hash value based on its children. The root is always kept secure; that is, it never leaves the chip. Moreover, any tamper with a counter leads to the failure of reconstructing the root. Depending on the tree structure, Merkle trees can be non-parallelizable (e.g., General Merkle Tree) or Parallelizable (e.g., SGX style counter tree) [99]. Since hashes in the general Merkle Tree style trees are calculated over the bottom level hashes, the tree update must be done sequentially. ToC integrity trees, on the other hand, can perform a parallel update of the tree, as the MAC values are not calculated over the

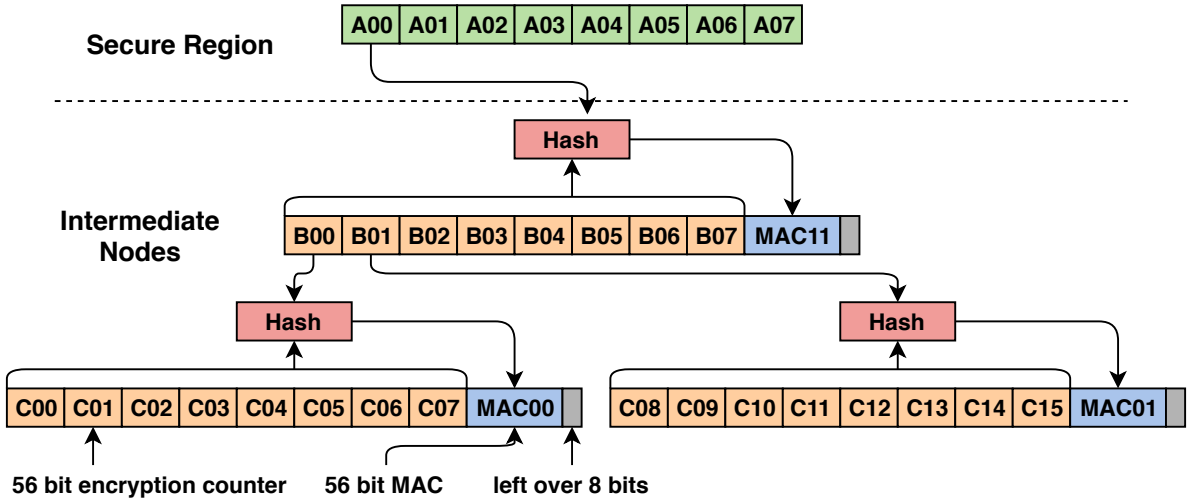

Figure 2.2: SGX style Parallelizable Merkle Tree

below level MAC value. Figure 2.2 illustrates the organization of ToC integrity tree where each node is comprised of eight counters. The MAC values are calculated over these eight counters and one counter from the parent node as in Figure 2.2.

Read and Verify

To better understand the verification step in ToC integrity trees, Figure 2.2 demonstrates a scenario of verifying counter `C00`. Note that `C00` falls within a block (64 bytes) that contains counters `C00 - C07` in addition to a MAC value. However, verifying `C00` also requires reading `B00` in the upper level, and then calculating the MAC value over `C00 - C07` and `B00`, then compare it with `MAC00`. However, it is important to note that this is assuming `B00` is already verified and cached in the processor chip. However, if `B00` is not present in the processor chip, it must be also verified the same way before we use it to verify `C00`.

Clearly, there is an inter-level dependency in the integrity tree, and missing an updated MAC due

to a crash can cause the whole recovery process to fail.

Write and Update

To better understand how updates propagate through the ToC integrity tree, let's take the case of updating C00. For now, let's assume that there is no expectation of integrity tree recovery after a crash. In its simplest form, updating (incrementing) C00 requires recalculating MAC00 after incrementing B00. Similarly, MAC11 will be recalculated with the incremented B00 and A00 values. One important aspect to note here is that on each update, the MAC values on the affected nodes can be calculated in parallel using the incremented counter values. In contrast, and for regular Merkle Tree, calculating the upper levels requires the MAC value as an input, hence mandating the serialization of updates (bottom-up). Thus, ToC trees provide parallelism in updating the tree mainly because calculating the values of counters affected on each node can occur in parallel, hence calculating the corresponding MAC values on each affected node.

ToC Advantages

ToC provides security advantages over the Merkle tree, since in the ToC each version is used in the calculation of two different MAC values, namely, the MAC value in the same node and the MAC value in the child node. This, in turn, makes it harder to perform replay attacks. In ToC integrity-protected systems, the attacker needs to replay an old intermediate version value that generates the same MAC value in the same node, and results in the same MAC in the child node. However, in Merkle tree, the attacker only needs to replay a counter value that generates the same MAC value of the counter. For example, as shown in Figure 2.2 if an attacker wants to replay an old value of B00, the value should generate the same MAC value as MAC11, and should result in a value that generates the same MAC as MAC00. Moreover, the new value of B00 should

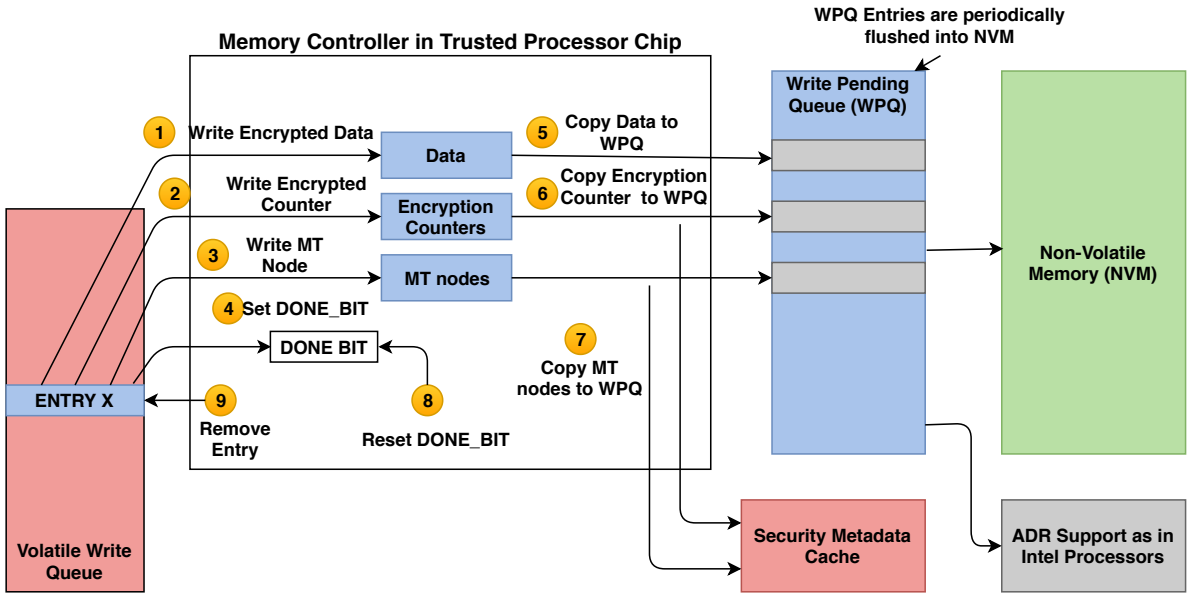

Figure 2.3: Atomic Persistence of Integrity-Protected NVMs.

generate the same value in A00, and the same process is repeated for all the levels of the ToC. In other words, to successfully perform an attack in a ToC integrity-protected system, the attacker needs to use a value that generates the same MAC values for all of the tree levels. On the other hand, for a Merkle tree integrity-protected system, the attacker only needs to use a value that generates the same MAC value in the parent node, then the parent MAC—which is basically the same, will be used to generate the upper levels. Moreover, ToC allows parallel calculation of upper levels updates, as the updates of upper levels does not depend on the MAC value of the child level. However, parallelism is not used when a lazy update scheme is used as we only write one cacheline and rely on eviction to propagate the updates to the upper levels.

Metadata Cache Update Scheme and Recoverability in Persistent Memories

Security metadata cache, caches the integrity tree nodes and encryption counters. The integrity tree can be eagerly or lazily updated. In eager update schemes, each write needs to update all the related nodes up until the root in the cache. Thus, the root always reflects the most recent state of the tree and can be used to verify the memory integrity after recovery. In contrast, the lazy update scheme, updates the leaf on each write and relies on propagating the updates upwardly only after the eviction of updated nodes. In the lazy update scheme, the root might still be stale while the metadata cache has the most recent values. Therefore, lazy update scheme is commonly used in systems with no expectation of recovery [19, 99].

In Merkle tree schemes where the tree can be regenerated using only the leaves (e.g., in General Merkle Tree), it requires a long time to rebuild the tree. Once the tree is reconstructed, the generated root will be compared against the one inside the processor chip which has been eagerly updated. In contrast, the lazy update scheme has no way to verify the integrity of the reconstructed tree since the root is out-of-date; the root does not reflect the most recent changes to memory before the crash. However, in the ToC integrity tree, it is impossible to regenerate the previous state of the tree from the leaf encryption counters since every intermediate node contains versions, the updated value of which could be lost during a crash. Due to such inter-dependency of levels, and the volatility of meta-data cache, it is very difficult to recover systems with ToC even if an eager update scheme is maintained.

While an eager update is suitable to rebuild the Merkle tree using only the encryption counters, it is not the case with ToC integrity tree. In ToC integrity tree, each node contains a MAC value calculated over the node counters and a nonce from the parent node. This inter-dependencies makes it very complex to retrieve the lost intermediate nodes during the recovery process. In lazily updated ToC integrity tree systems, the root is not enough to verify the integrity of the memory as

it might be stale. Thus, to verify the integrity of the memory the integrity tree should be restored, while each node is used to verify the integrity of lower and upper levels.

Counter Recovery Schemes

Recovering encryption counters, the leaves of Merkle Tree, is generally considered the first step in recovering the tree. Prior work on general Merkle Tree [19,90] explored how to recover encryption counters after a crash. One solution, Osiris [90], relies on encrypting Error-Correcting Code (ECC) written with data. By limiting the number of updates to a counter before persisting it to memory, e.g., every 4th write, it can recover the counter used to encrypt the data by relying on the fact that a large number of errors will be detected by ECC when a wrong counter is used. By trying multiple counter values, Osiris can recover the counter used to encrypt the data. For more details on Osiris, the reader is referred to [90].

While Osiris presents a novel approach that reduces the overhead of persisting counters significantly, there are many other competing approaches [55]. For instance, as also discussed in [90], part of the encryption counter used for encryption can be also written with the data and thus strict the persistence of the whole encryption counter can be relaxed. For the rest of this chapter, we assume Osiris can be used, however, any other counter recovery scheme would work.

ToC Recovery Schemes

The state-of-the-art scheme, Anubis [99], is the first to enable recovering ToC trees without strictly persisting all tree updates to the memory. Anubis mainly builds upon the observation that it is sufficient to just recover the state of the cache before a crash, even if a lazy cache update scheme was used. In other words, if we can recover the content of the metadata cache after a crash, to

the same content before the crash and verify it, then it is sufficient. To do the cache recovery part, Anubis persists security metadata cache updates in the memory in a region called the shadow region, i.e., writing cache updates additionally to memory. However, to fit the tag of the cache block and the block content into 64-byte cache lines, part of the counters (i.e., most-significant bits) are trimmed and their updates are persisted immediately. Meanwhile, to verify the integrity of the shadow cache after a crash, a small integrity tree protects the shadow region and uses eagerly updated Merkle tree. The root of the shadow region tree is updated on each cache update. However, the intermediate nodes do not need to be persisted; the shadow region tree is implemented using a general tree where the root can be generated from the leaves. Thus, after power restoration, Anubis reads the shadow region and then calculates the root of the shadow region and compares it with that inside the processor chip. Note that in Anubis there will be two roots inside the processor chip, one for the shadow region tree (regular tree) and the other one (ToC tree) is for the rest of the memory. Moreover, Anubis keeps two integrity trees, one for the memory which is the ToC integrity tree, which is updated using the lazy update scheme, whereas the second one is the small general Merkle tree covering the shadow region, which is updated using eager update scheme.

Obviously, Anubis incurs almost double the write bandwidth: on each memory write, updating metadata cache needs to be persisted to memory.

Atomic Update of Security Metadata

While persisting the security metadata allows the system to recover after a crash, if the crash happens when the security metadata and data could not be both persisted, it will lead the NVM content to be inconsistent. To ensure the security metadata is consistent with the data, the update should be done atomically. Modern processors provide enough power to flush the content of the Write Pending Queue (WPQ) when a crash occurs, and the power to flush the WPQ content is

Table 2.1: Schemes comparison

Solution	**ToC Recoverability**	**Low Recovery Time**	**Performance Overhead**	**Write Overhead**
Osiris [90]	X	Not Applicable	Not Applicable	Not Applicable
Strict (Triad-NVM [19])	✓	✓	X (very high)	X (very high)
Anubis	✓	✓	X (high)	X (high)
Phoenix	✓	✓	✓(low)	✓(low)

provided by the Asynchronous DRAM Refresh (ADR) feature [19]. Therefore, all writes that have reached the WPQ are considered to be persisted. Additional bits, such as READY_BIT or DONE_BIT can be used to ensure the content of persistent registers are inserted atomically to the WPQ [19, 55]. Figure 2.3 shows how atomic updates are done, the encrypted data, encryption counter, and the updated MT nodes are moved to persistent registers, then the DONE_BIT is set. After that the updates are moved atomically to the WPQ, then the data is written to the NVM, and finally the DONE_BIT is reset and the entry is removed from the volatile WPQ.

Motivation

The main goal of Phoenix is to provide a practical solution for recovering ToC integrity trees. Table 5.1 summarizes the features of the current solutions in contrast with Phoenix. While Osiris [90] provides an efficient counter (tree leaves) recovery scheme, it fails to recover the ToC integrity tree. As discussed earlier, ToC instances have inter-level dependence, which makes rebuilding a ToC from the recovered leaves impossible. Strict persistence schemes require persisting all tree updates in memory while using an eager update scheme, i.e., the root reflects the most-recent tree status. Thus, the strict persistence has a low recovery time, where there is no need to rebuild tree, although incurs a significant write overhead and result in performance degradation.

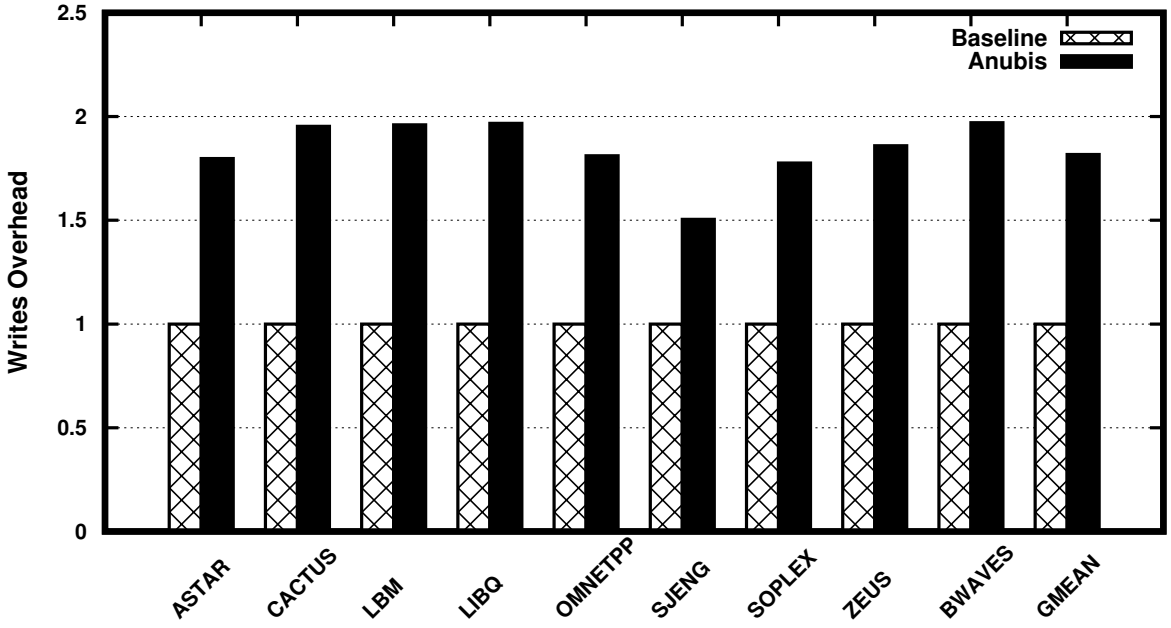

Figure 2.4: Anubis extra writes.

For instance, for an 8TB memory system, strict persistence needs to persist additional 13 writes on each regular memory write, i.e., reducing NVM lifetime and increasing write bandwidth by 13x. Clearly, strict persistence is impractical. Anubis brings down the overhead of strict persistence significantly, although it is still high. In particular, Anubis incurs 2x the write bandwidth by persisting each update to cache in the shadow region. Thus, Anubis reduces the lifetime of NVM systems to almost half of its actual lifetime, although the lifetime of NVMs is already short, to begin with. Moreover, NVM writes are slow and power hungry, hence can significantly degrade the performance and increase the overall power consumption. Figure 3.3 shows the overhead of Anubis scheme, which can limit its deployment, and motivates for this work. In particular, Figure 3.3 shows the impact of Anubis on the number of writes. On average Anubis, incurs almost 2x the number of writes and average performance overhead of 7.9% compared to baseline secure NVM without recovery support. The goal of Phoenix is to provide an NVM-friendly solution that does

not incur significant NVM writes. Thus, Phoenix is proposed as a practical solution that realizes low-overhead secure and recoverable NVMs.

Phoenix Design

Before delving into the details of Phoenix's design, we first discuss the main goals and design principles that inspire Phoenix. The goal of Phoenix is to enable recovery of ToC integrity trees with an emphasis on low write overhead. We realize that any practical solution proposed for NVMs must have low write overhead. Thus, Phoenix mainly aims for ultra-low write overhead while still enabling recovery of ToC integrity trees. The first observation that Phoenix builds upon is that recovery of ToC integrity trees can be achieved by recovering the lost content of security metadata cache. While this observation has been also made in prior work, e.g., Anubis [99], enabling such a recovery of cache content has been done in a way similar to write-through, by persisting the writes made to security metadata cache into a shadow region in the NVM, which has been proven to be very expensive when used with NVMs [90]. However, Phoenix is based on the fact that we can actually recover the cache content without exact/accurate shadowing of all of its content. In this chapter, we make a novel observation and contribution that we can securely recover the lost cache contents and verify them while still relaxing the shadowing operation.

In particular, we observe that recovering ToC integrity trees by relying on restoring the cache content before a crash has two major requirements. First, there must be a mechanism to verify that we recovered the most recent cache content before a crash and its contents have not been tampered with. Second, the root of the Merkle Tree must reflect the updates of all memory including the cache contents just before a crash. By ensuring these two requirements are satisfied, the security metadata cache can be recovered and the rest of the memory verification is verified through a Merkle Tree on each memory access. In other words, simply bringing the metadata cache and

Merkle Tree root (unaffected) to the state before a crash is sufficient to ensure crash consistency of security metadata.

Prior work, Anubis [99], achieved such a cache recovery mechanism by relying on a reserved region in the NVM called the shadow cache, in which any updates of a lazy-update ToC metadata cache is copied, thus resulting in doubling the writes. To ensure the integrity of the shadow cache, Anubis applies a small Merkle Tree over the shadow cache while keeping its root in the processor and following an eager update scheme. After a crash, the cache content can be restored from the shadow region and its integrity can be verified using the small Merkle Tree (the eagerly updated one), which also has its root kept in an NVM (or NVM-backed) register inside the processor chip. On the other hand, Phoenix is mainly based on the fact that most updates to metadata cache in the lazy-update scheme are for leaves. However, shadowing leaves updates to memory might be unnecessary if we can have the following: ① a mechanism to verify the most-recent cache state including leaves but without necessarily shadowing them, and ② the ability to recover leave updates.

Phoenix employs state-of-the-art counter recovery schemes, e.g., Osiris and phase-based recovery [90], to relax updates to the shadow region in the cache while simultaneously allowing to recover the exact content of the cache right before a crash. Specifically, Phoenix selectively decides which security metadata should be shadowed strictly and which ones can be relaxed. Even though it relaxes the shadow region update, Phoenix enables the reconstruction of the cache content (including relaxed leaves) and allows the verification of recovering the exact content before a crash. Since most updates to the security metadata cache are caused by leaves updates, Phoenix is expected to significantly reduce the number of writes while allowing fast recovery of ToC trees. The main downside of Phoenix is that it requires additional work before reconstructing the lost cache content and verifying it.

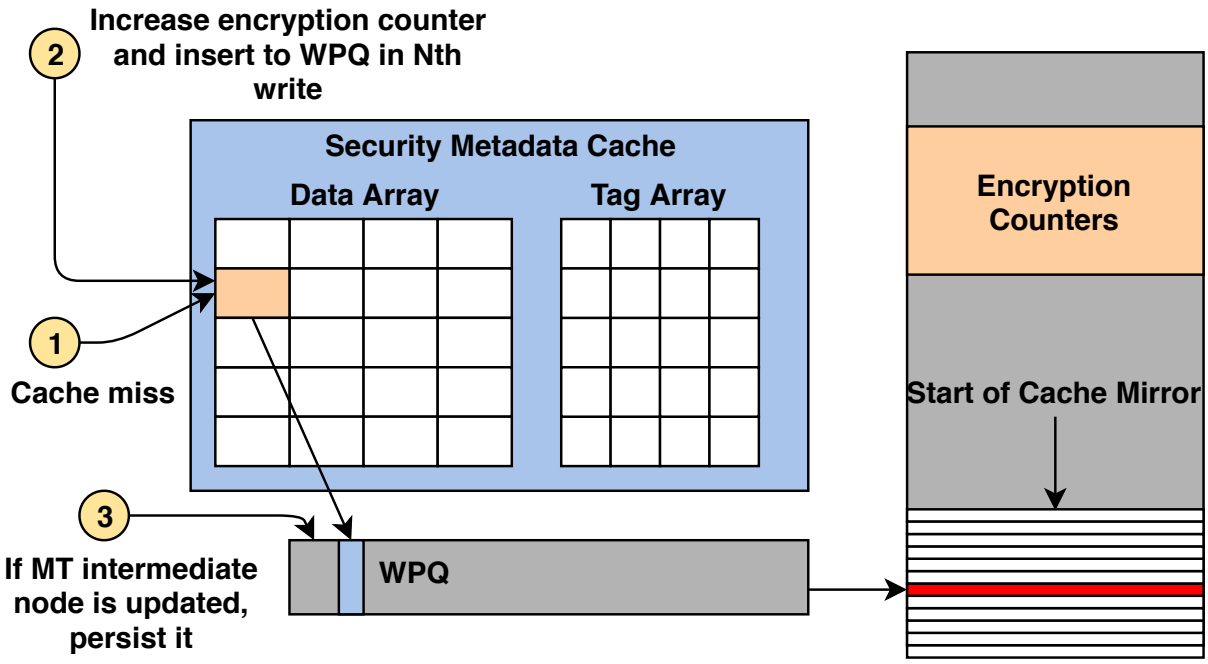

Figure 2.5: Updates Tracking

The rest of this section discusses the design details of Phoenix. Note that the main distinction between Phoenix and other schemes lies in Phoenix's ability to recover the exact cache content before a crash, although without a strict shadowing of cache content to NVM. Phoenix is the first scheme to enable lazy-update cache recovery without the need to persist each security metadata cache update.

Selective Persistence

Upon a crash, the cache loses its contents. Losing the cached security metadata results in integrity verification failure, thus the cached security metadata needs to be persisted.

Strictly persisting the security metadata incurs tens of additional writes, and to avoid those unnec-

essary extra writes we opt for persisting only the unrecoverable nodes of the metadata. Security metadata contains the encryption counters and the Merkle tree nodes, while ToC integrity tree nodes are composed of 8 counters and a MAC calculated over the 8 counters and the parent of the node. Since the encryption counters can be retrieved without strictly persisting them using Osiris [90], we are going to follow a similar scheme by persisting the encryption counters with every N-th write, or on eviction. On the other hand, the intermediate ToC nodes are not recoverable, therefore we suggest persisting these cached nodes to successfully recover from the crash. To achieve that, we allocate a small region in the NVM which is the same size of the security metadata cache (about 256 kB) which we refer to as the Cache Mirror (CM). Whenever an intermediate ToC node is written in the cache, this update will be persisted and its address will be copied to the CM, while updating the encryption counters with every N-th write. Figure 2.5 shows how selective persistence is done, whenever a write happens, if the write resulted in updating an intermediate node, the updated intermediate node is copied to the CM region. During the recovery process, the contents of the CM are used to recover the lost cache contents and refresh the ToC to ensure a secure recovery process. Since the security region is defined by the boundaries of the processor, the integrity of the CM should be guaranteed before it can be used during the recovery. Thus, we apply a small Merkle Tree (MT), four levels with an arity of eight, over the CM while keeping the root of this tree in the processor. During the recovery, the integrity of the CM region is verified by building the CM-MT and comparing the resulting root with the processor kept root.

Phoenix Operation

Phoenix read operation is merely a read and verify operation, and does not require any changes or special handling. In particular, the read operation in Phoenix does not modify the security metadata cache except for eviction, which is discussed in subsection 2. On the other hand, the write operation results in an encryption counter increment to ensure a new encryption pad for the

modified block. The encryption counter increment will not affect the MAC value in the node nor increment the parent as we are using a lazy update scheme, but Phoenix will be triggered and the address of the modified counter will be copied to the CM. However, when an encryption counter block is evicted the parent node should be fetched and both nodes should be updated. Despite using a lazy update scheme, it is important to persist the encryption counter at every N-th (4th in Phoenix) write, or on eviction to enable encryption counter recovery. It is important to keep in mind that updates of ToC node and the data are to be done atomically using the Write Pending Queue (WPQ) and a ready bit as described earlier. While encryption counters are updated at every N-th write, ToC intermediate nodes need to be persisted each time they are modified, thus the addresses of the intermediate nodes are copied to the CM, and the intermediate nodes are persisted into the NVM.

Phoenix+ Operation

While Phoenix persists intermediate nodes on each update, and persists encryption counters on N-th update or eviction. Phoenix+ relaxes persisting encryption counters on eviction, to only persist encryption counters on the N-th write. By doing this, Phoenix+ reduces the number of writes and the performance overhead significantly. Phoenix+ relies on recovering the encryption counters while working, by utilizing the encryption counters recovery scheme. Notice that, recovering the encryption counters on the run might add performance overhead if done in a sequential manner, but we assume N-ECC engines (4 in our design) to retrieve the latest value of the used encryption counter. Keep in mind, that evicting an encryption counter does not update its value, but affects its parent, and still affect the encrypted data. Thus, the old encryption counter value integrity can be recovered and verified using the parent value.

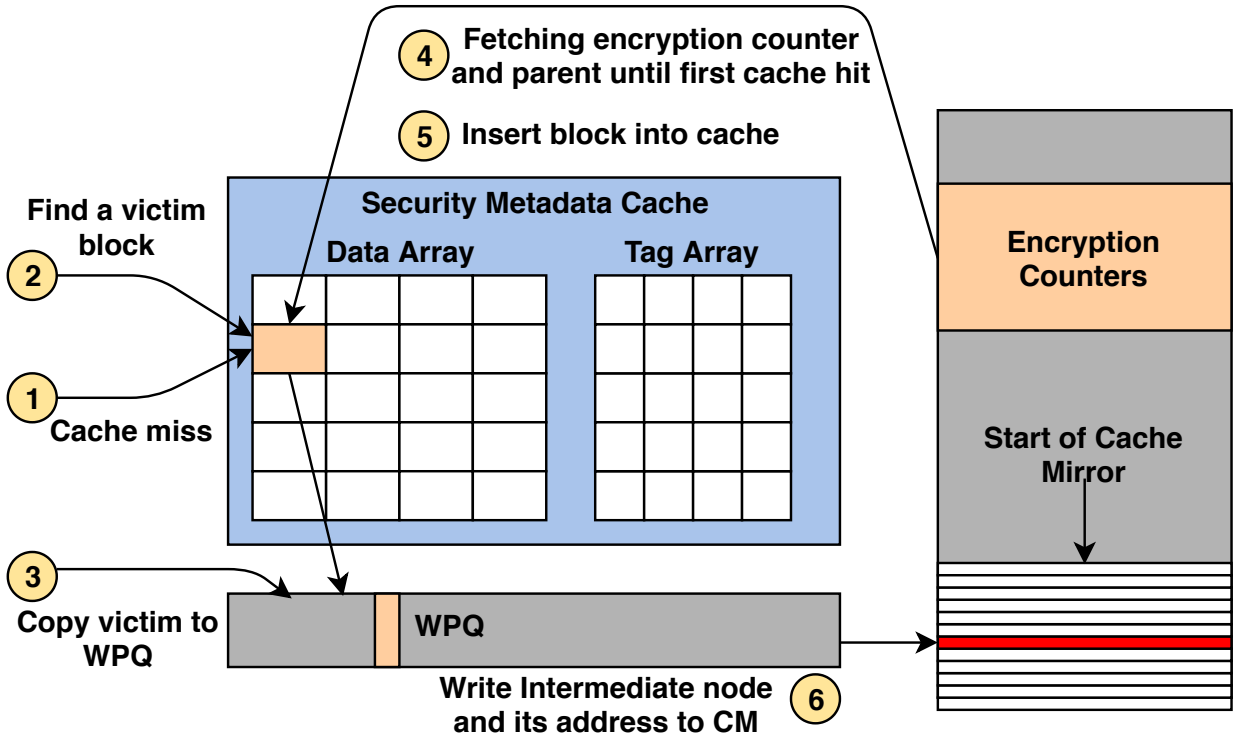

Figure 2.6: Eviction

Eviction

The lazy update scheme we use in Phoenix reduces the number of writes while relying on eviction to propagate the nodes update. Figure 2.6 shows the eviction process. In the case of a encryption counters cache miss, the memory controller selects a victim block to be evicted from the cache using the Least Recently Used (LRU) replacement policy. The victim block is then inserted to the WPQ in case it was an intermediate tree node. Note that we are not persisting the encryption counter on eviction, since we are relying on Osiris to retrieve the encryption counter's most recent value while running. Persisting the encryption counter on eviction will improve the performance slightly, as we can immediately use the fetched encryption counter after verifying its integrity,

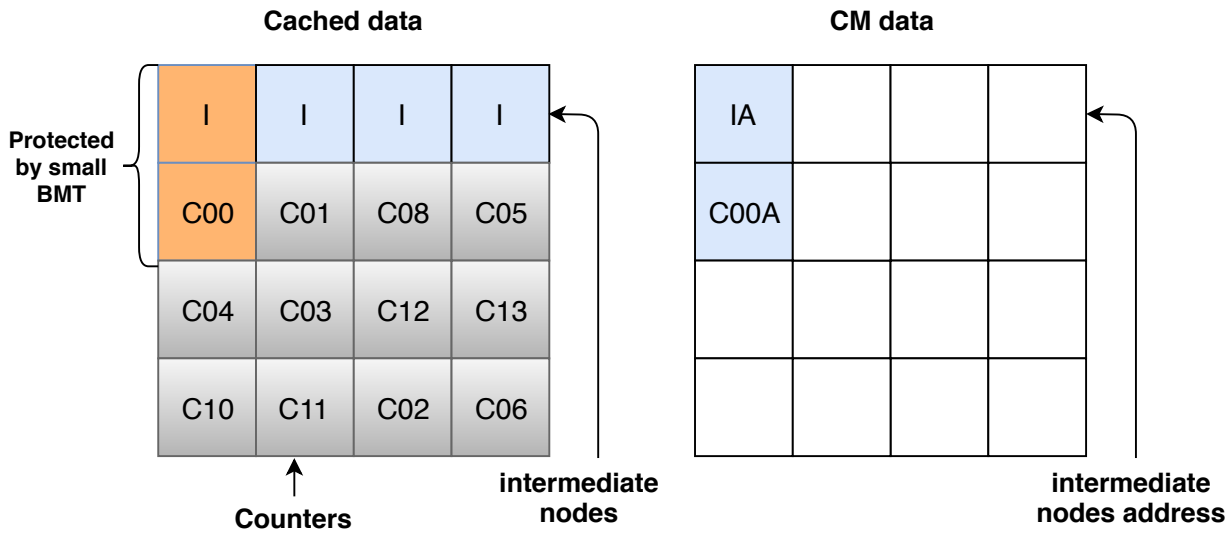

Figure 2.7: Cache Mirror

although will increase the number of writes to the NVM. Since our goal is to successfully recover the ToC while maintaining a low number of writes, we opt for recovering the encryption counter while running by only persisting the encryption counter at every N-th write. To ensure the data consistency we assume the evicted encryption counter, intermediate ToC node, data, and CM data are inserted atomically to the WPQ as described in section 2.

Imprecise Cache Mirror

The Cache Mirror (CM) region, shown in Figure 2.7, is a small reserved region in the NVM. The CM only contains the addresses of the dirty intermediate nodes and the addresses of the dirty counters, while the actual dirty intermediate nodes are persisted to their actual locations. The CM contents are used to securely recover the system after a crash. To ensure the integrity of recovery, the dirty cached intermediate nodes and the dirty encryption counters are protected with a small

general MT. This small MT that only covers the dirty intermediate nodes and dirty encryption counters is eagerly updated, and its root is always kept in the processor. Notice that by relaxing the CM contents to contain only the addresses of dirty intermediate nodes and the addresses of dirty encryption counters, we were able to drop the number of writes significantly. Moreover, using a small MT to cover only the dirty intermediate nodes and dirty encryption counters, we were able to drop the performance overhead. We note that when a lazy update scheme is used, the root is no longer suitable as a single point of memory content integrity verification. As a matter of fact, the cache contents are the most updated nodes, and the nodes are used to verify the integrity of fetched nodes. When a crash happens, and the cached nodes are lost. However, we can recover the leaf nodes using encryption counters recovery, although the integrity of these nodes needs to be verified. The parents of these nodes can be either up-to-date in the NVM, or cached nodes lost during the crash. Thus, we make sure to persist the intermediate nodes, and use the small MT root to ensure the integrity of cached intermediate nodes.

Integrity Verification

The secure region is defined by the processors boundary. While on-chip memory is considered secure, that is not the case with NVM. Thus, whenever a data block is fetched from the NVM its integrity needs to be verified. To verify the integrity of any block, its parent needs to be fetched and used to calculate the MAC value of the verified block. However, once the parent is fetched, its integrity needs to be verified which will result in a recursive operation until the first parent cache hit. Once one parent is found in the cache, its integrity is considered to be verified, and is used to calculate the MAC of the child node. If the calculated MAC value matches the child node's stored MAC value, the child's integrity is considered verified. For the CM region, since its size is very limited (256 kB) it is more suitable to use an eagerly updated MT and store its root in the processor. Using an eagerly updating scheme means the root always reflects the most recent tree state. The

CM MT is four levels using 8-ary tree, thus it is feasible to rebuild the tree during recovery and compare with the stored root to verify its integrity.

Recovery

The recovery process starts by loading the pre-crach cached intermediate nodes from the NVM, using the addresses saved in the CM region. Then, the integrity of the loaded intermediate nodes is verified using the small MT root. When the intermediate nodes are verified, any interrupted write operation is resumed, by checking the DONE_BIT and completing the pending operations to successfully complete the atomic write. Notice that the small MT root is eagerly updated, and always kept in the processor. Moreover, the small MT root is calculated over the dirty cached intermediate nodes, thus its update is infrequent, since most of the updates are done to the leaf nodes. In turn, the overhead of eagerly updating the small MT root is negligible. Note that we are restoring the encryption counter during the normal operation, and the encryption counters are not persisted nor recovered during the recovery process, since they are recovered when fetched.

Once the CM integrity is verified, the DONE_BIT is checked and any pending write operations that were in the persistent registers before the crash are moved to the WPQ and executed. After the pending write operations are executed, the CM contents are used to restore the cached intermediate ToC nodes. While the intermediate ToC nodes are ensured to be recovered to the most recent state using the CM, the encryption counters are not. To restore the encryption counter to the most recent state, we use the CM contents to retrieve the addresses of the cached encryption counters, then fetch the counters and use Osiris to retrieve the most recent counter value. After the encryption counters are updated to the most recent values, the cache is restored to its previous state before the crash, and its integrity can be verified using the small MT root. Notice that in case of the CM region is tampered with, and the calculated root of the CM region does not match the stored root

Algorithm 1 Phoenix Recover Algorithm

```
1: Read CM
2: for Nodei in CM do
3:     Nodei ←Memory[Nodei]
4:     if Nodei is EncryptionCounter then
5:         Nodei ←Osiris[Nodei]
6:     end if
7:     MetadataCache ←Nodei
8: end for
9: CMRoot_Recovered ←GenerateRoot[MetadataCache]
10: if CMRoot_Recovered = CMRoot_Stored then
11:     WPQ ←PersistentRegisters
12:     Memory ←WPQ
13:     ContinueNormalOperation
14: else
15:     TheSystemisUnrecoverable
16: end if
```

in the processor, the recovery process fails and the integrity of the NVM is declared unverifiable.

Security Discussion

In traditional persistent secure systems, the security of the data is protected using the counter mode encryption, and the integrity of the encryption counters is protected with MT. The root of the MT is always kept in the processor, and memory content's integrity is verified by calculating the root and comparing it with the processor stored root. This scheme works well for eagerly updated MT, which is not the case in our scheme. Phoenix+ scheme relies on a lazy update scheme, which means whenever a leaf counter is updated we do not update the parent of the counter, nor update the associated MAC with the leaf counter node, but rely on the N-th write to the same counter to propagate the update. In Figure 2.2, if the counter `C01` is updated twice and then got evicted, the parent node `B00` will be updated, and the MAC value `MAC00` will be stale. Notice that even the counter `B00` will not be stale in the NVM, so the next time counter `C01` is fetched it still can

be verified successfully using the stale MAC00 and the parent B00, and its most recent value can be recovered using Osiris [90]. In the lazy update scheme, the root of the ToC can be stale, and the most updated state is preserved in the cached intermediate nodes of the ToC. In the recovery process, it is essential to guarantee the integrity of the CM region as it reflects the most recent state of the tree. Thus, the integrity of the CM is protected using a small BMT and the root is eagerly updated and kept in the secure region (processor).

Methodology

Evaluating our scheme was done using Gem5 simulator [22], a cycle-level simulator. Table 5.2 shows the used configuration, we simulate a 4-core X86 processor with 16GB PCM-based Main Memory with parameters modeled as in [49]. The evaluation was done by running 9 applications from the SPEC 2006 benchmark suite [37]. The used benchmarks include memory intensive applications in both read and write intensive applications. For each application, we simulate 500M instructions after fast forwarding to a representative region.

In our evaluation, we model the integrity-protection using ToC, encryption aspects, security metadata cache, hash calculation latency, and cache mirror region integrity protection.

Evaluation

In this section, we evaluate our scheme based on the `Write Back scheme` as the baseline, and compare it with state-of-the-art scheme, Anubis [99]. We evaluate the additional number of write incurred by each scheme, the performance of Phoenix and Phoenix+ schemes, then we show the sensitivity to cache size and recovery time.

Table 2.2: Configuration of the simulated system

Processor	
CPU	4 Cores, X86-64, Out-of-Order, 1.00GHz
L1 Cache	Private, 2 Cycles, 32KB,2-Way
L2 Cache	Private, 20 Cycles, 512KB, 8-Way
L3 Cache	Shared, 32 Cycles, 8MB, 64-Way
Cacheline Size	64Byte
DDR-based PCM Main Memory	
Capacity	16GB
PCM Latencies	Read 60ns, Write 150ns
Encryption Parameters	
Security Metadata Cache	256KB, 8-Way, 64B Block
CM in Phoenix	256KB
CM in Phoenix+	256KB
Persistence Limit	4

Phoenix Writes

To evaluate our scheme, we compared the number of writes incurred for each of the following schemes.

1) **Write Back (Baseline)**: This is a simple ToC integrity tree scheme with write back. This system only writes on eviction and does not provide any recoverability.

2) **Anubis SGX**: The Anubis scheme for ToC integrity tree updates the ToC lazily and writes all the ToC updates to a shadow region.

3) **Phoenix**: This scheme updates the ToC lazily while persisting the updates for intermediate ToC nodes, and relaxes the updates for leaf nodes until eviction or the counter is written N times.

4) **Phoenix+**: This scheme reduces the number of writes in Phoenix by only persisting the leaf nodes on the Nth write, and relies on a counter recovery scheme (Osiris [90]) to recover the counters on the run.

Figure 2.8 shows the number of writes incurred by the above schemes. Considering the Write

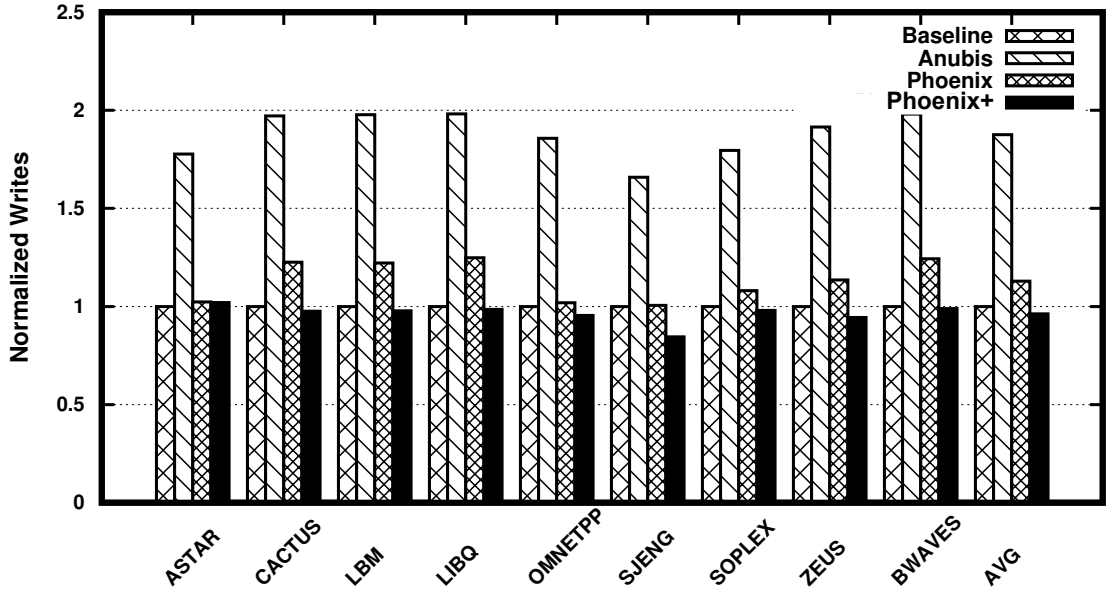

Figure 2.8: Phoenix Extra Writes

Back as the baseline scheme, we notice that Anubis incurs an average of 87% extra writes, while Phoenix incurs an average of 12.9% extra writes, and Phoenix+ reduces the writes to less than the write-back by an average of 3.8%. Phoenix+ reduces the number of writes to less than Write Back scheme while achieving the recoverability of ToC. Phoenix+ achieves this reduction by utilizing the lazy update scheme for the ToC, and by eliminating the eviction writes for the encryption counter nodes, while using Osiris counter recovery to recover the latest value of the encryption counter each time it is fetched.

Phoenix Performance

To evaluate Phoenix, we model and compare the aforementioned four schemes. Figure 2.9 illustrates Phoenix's performance in comparison to other schemes. Considering the Write Back scheme as the baseline, Anubis provides the ability to recover the ToC with 7.9% extra performance over-

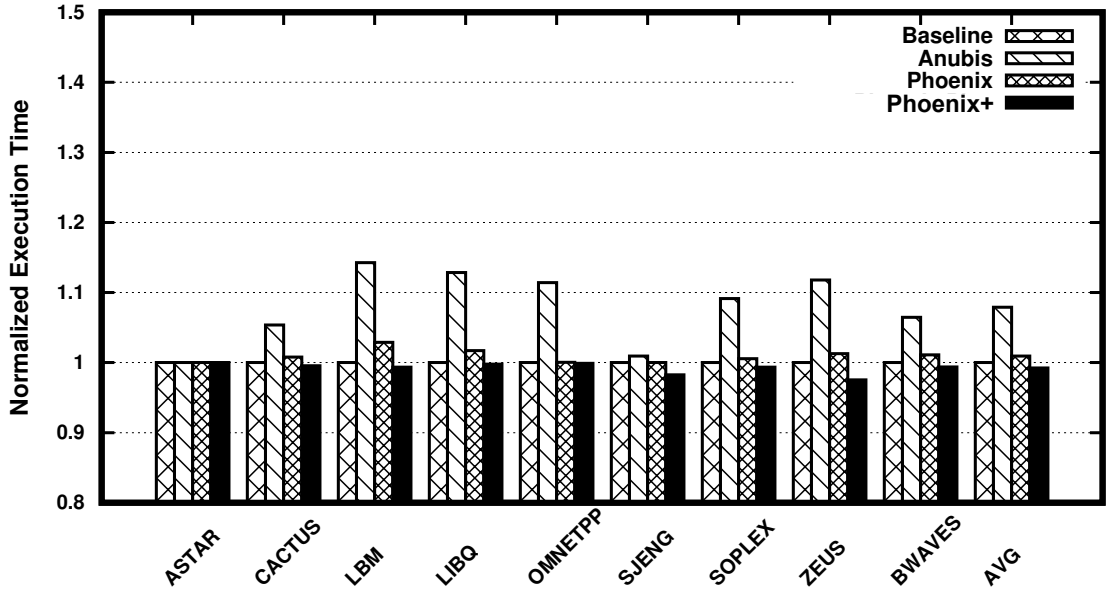

Figure 2.9: Phoenix Performance

head. Phoenix is not only capable of recovering the ToC, but also achieves a performance of 0.8% better than the write back scheme. That is, Phoenix+' performance is better than the Write Back scheme, thus Phoenix+ reduces the overhead by 8.7% compared to Anubis. For instance, we notice, also from Figure 2.9, that Phoenix in both versions is performing better than the baseline for *CACTUS* benchmark. Moreover, using memory intensive benchmarks shows that Phoenix+ performs slightly better than the Write Back scheme, while this difference is expected to be more noticeable with less memory intensive applications. Phoenix reduces the overhead by relying on lazily updating the ToC while persisting each update to the intermediate ToC nodes. Notice that relying on the lazy update reduces the frequency of updating the intermediate nodes until the leaf node is evicted. On the other hand, Phoenix+ takes one step further to relax persisting the leaf counters on eviction and relies on Osiris as a counter recovery scheme to recover the counters while running.

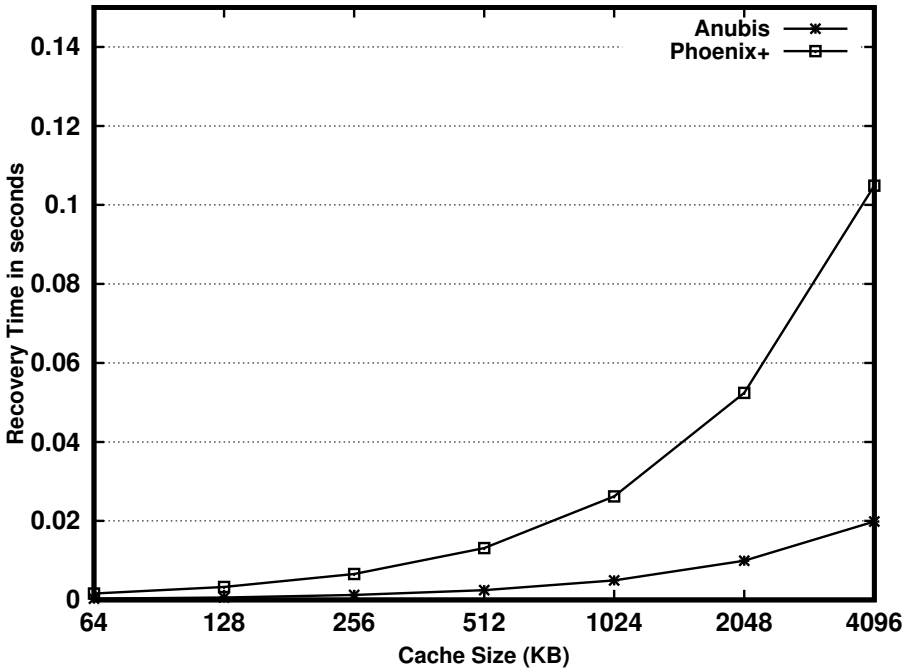

Figure 2.10: Recovery Time

Sensitivity Study

Recovery Time

System recovery of ToC protected systems was not possible except for strict persisting scheme, until recently. Anubis [99], Phoenix, and Phoenix+ allow the recovery in less than a second, due to making the recovery time a function of the cache size instead of the memory size. While Anubis [99] relies on a lazy strict persistent scheme which results in extra 87% extra writes to achieve the recoverability of the ToC integrity protected systems in the same time that Phoenix requires to recover the same NVM. Figure 2.10 shows the recovery time of Anubis and Phoenix+ regarding the cache size. Figure 2.10 shows that both schemes achieve a recovery time of less than a second, even for extremely large cache size (4MB) Phoenix recovery time is $\approx$0.12 seconds for Phoenix+

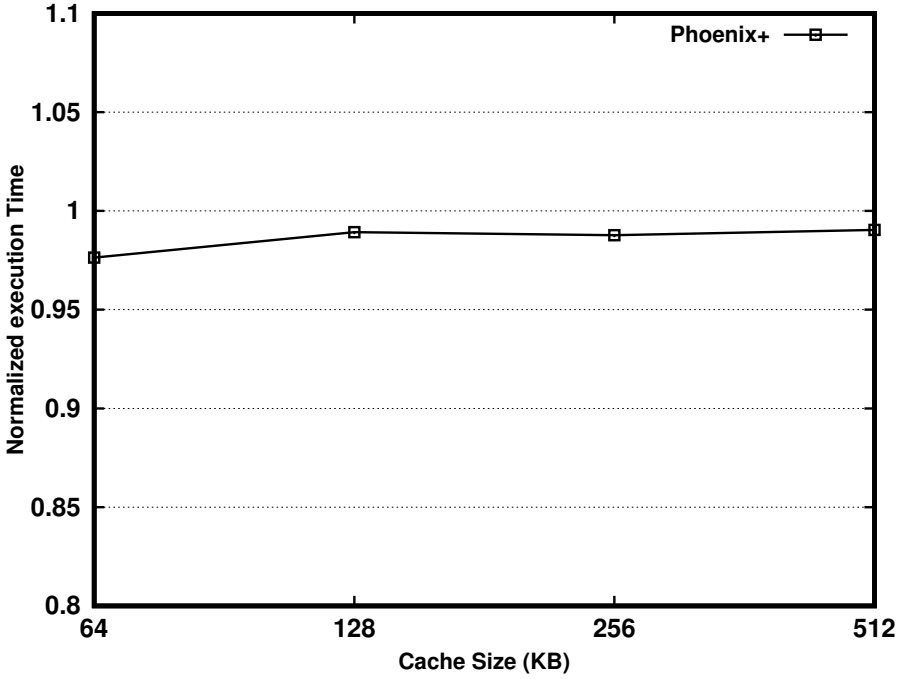

Figure 2.11: Sensitivity to Cache Size

and ≈0.015 seconds for Anubis. Notice that Phoenix+ recovery time is higher than Anubis scheme as it requires retrieving the leaf counters values during the recovery process. For this sensitivity test, the worst case scenario is considered to calculate the recovery time, by considering all the eight counters in each leaf node are stale. We notice that Phoenix+ trades a very small amount of recovery time for reducing performance overhead and the number of writes of the system.

Performance Sensitivity to Cache Size

Phoenix+ allows the recovery of ToC integrity protected NVM as a function of the cache size. To fully evaluate the scheme, we vary the cache size and measure the performance overhead of our scheme. As shown in Figure 2.11, the performance of Phoenix+ almost stays the same. This can be explained by Phoenix operation; Phoenix+ performs in a manner similar to the baseline (Write

Back). However, Phoenix+' performance depends on the number of writes to cached data: the more writes to the cached data will result in more counter writes, which results in more Phoenix writes.

Counter Persistence Limit

The number of writes on which the encryption counter is persisted clearly affects the performance of Phoenix. Using a large number of writes before the encryption counter is persisted reduces the number of writes and the performance overhead. However, this comes at a cost: a large persistency limit would cause higher recovery time and higher performance overhead as the counter latest value needs to be recovered each time its fetched or during recovery. The performance overhead can be avoided by using multiple ECC engines to recover the counter value. In our design, we opt for using the 4th write to be the persistence limit, choosing to persist the counter at the 4th write provides a very low performance overhead and enables the recovery within less than a second.

Related Work

The most related work to Phoenix are Anubis [99], Triad-NVM [19], Osiris [90], and Crash Consistency [55]. Anubis [99] addresses the recovery time of NVM systems and uses a shadow region to track down all the changes to cache contents, where each writes to the cache results in a write to the shadow region. The shadow region facilitates recovering the cache contents in ultra-low time, but incurs 87% extra writes for ToC. Triad-NVM [19], on the other hand, discusses the trade-off between recovery time and performance, and reduces the recovery time by persisting N levels of the MT. On the downside, Triad-NVM does not work with ToC, and requires persisting multiple levels of the integrity tree.

To recover counters, Osiris [90] relies on the ECC-bits as a sanity check for the used encryption counter. By applying a stop-loss mechanism, Osiris restores the encryption counters using a limited number of trials. Osiris works for retrieving the encryption counters while assuming building the integrity tree is possible, which is not the case with ToC integrity tree. Crash Consistency [55] for counters recovery proposes an API for programmers to selectively persist counters, and ensures atomicity through a write queue and Ready-Bit. In order to reduce the overhead, it proposes selective counter atomicity of the persistent applications. The scheme depends on the amount of applications persistent data and does not address the recoverability issue of ToC.

There are several state-of-the-art works done in NVM security and persistence [28,45,68,76,77,81, 96, 100] without considering the crash-consistency and recovery that discusses to optimize the run-time overhead of implementing security to NVM. Most works employ counter-mode encryption for encrypting data and MT for ensuring integrity. However, to the best of our knowledge, none of the works consider the recovery and crash-consistency of integrity protected systems. As a matter of fact, any work that does encryption counters compression or increases the integrity tree arity boosts our scheme, by increasing the cacheability of the encryption counters and reducing the number of intermediate nodes. SecPM [100] proposes a write-through mechanism for the counter cache that tries to combine multiple updates of counters to a single write to memory, however, does not ensure recovery for ToC and incurs significant recovery time as in Osiris. While Anubis [99] discusses the reduction of recovery time of secure non-volatile memory and recovery mechanism for ToC, however; the scheme incurs almost 2x extra writes which reduces the NVM lifetime by half.

Conclusion

Phoenix is based on four observations, first, most updates of the lazily updated ToC are done to leaf nodes. Second, leaf nodes are the least likely to be evicted as they will be reused frequently for verification and update purposes. Third, leaf nodes can be recovered using any encryption counter recovery scheme, we used Osiris in our work, but any other scheme should work. Fourth, cached intermediate nodes can be persisted at their location instead of being copied to the shadow region, and the small MT only needs to cover the dirty cached intermediate nodes and the dirty encryption counters. Phoenix achieves recoverability with ultra-low recovery time while keeping the number of writes to the minimum in ToC integrity protected NVMs. Our solution achieves a significant improvement in the number of writes as it reduces the number of writes by 90.8% less than state-of-the-art scheme Anubis, and 3.8% less than the write back scheme, with a recovery time of less than a second in ToC integrity protected systems. In addition, Phoenix recovery time and extra writes are a function of the cache size, as it works by recovering the lost cached ToC nodes. In summary, Phoenix recovers the ToC in less than a second, reduces the number of writes significantly, and improves the performance.

CHAPTER 3: IMPROVING THE PERFORMANCE OF PERSISTENT APPLICATIONS IN HYBRID MAIN MEMORY

Background

In this Section, we discuss the most related topics to our work to help the reader understand our work, followed by the motivation of this work.

Emerging Non-Volatile Memories

Emerging NVMs such as 3D XPoint and Intel's Optane DC feature higher density, byte addressability, lower cost per bit, lower idle power consumption than DRAM, and non-volatility, but have higher access latency and limited write endurance [19,49,50,53]. Due to the non-volatility feature, they can be used as a storage to host filesystem, or as a memory either persistent or non-persistent. For instance, NVM-based DIMMs can be used to hold files and memory pages, which can be accessed using regular load/store operations. To realize this type of accesses, recent operating systems (OSes) started to support configuring the memory as persistent or conventional non-persistent through the DAX filesystems [75]. In DAX filesystems, a file can be directly memory-mapped and accessed using regular load/store operations without copying its content to the page cache [19]. However, NVM's access latency is 3-4x slower than the DRAM's access latency. Therefore, researchers proposed to build memory systems that have both NVM and DRAM portions [67, 95].

Hybrid Main Memory (HMM)

Hybrid main memory (HMM) systems are expected to have a large NVM portion due to its density and ultra-low idle power, and a small DRAM portion due to its fast read/write operations. HMM can be deployed in two different schemes, *horizontally* or *vertically*. In the vertical scheme, the NVM is connected as a new memory tier and the DRAM is used to cache the NVM's data [8, 10]. This scheme allows faster access to the large memory pool (NVM), and requires a special hardware to migrate data from the NVM to the DRAM, e.g., the caching of cachelines is handled by Intel's Xeon scalable processor's memory controller in Intel's Optane DC memory mode. However, such a scheme does not provide persistency due to the DRAM's volatility. In the second approach, a horizontal implementation of the HMM system exposes both the NVM and the DRAM to the physical address space, as in NVDIMM-P and Optane DC's application direct mode, and relies on the OS to handle data accesses and page migrations if required [41, 65, 67, 89]. In both cases, a hybrid memory management scheme is required to manage different persistency and performance requirements.

Different hybrid memory management schemes have been proposed in the literature based on the memory hierarchy. Schemes such as HetroOs [41], RTHMS [62], and Nimble [89] proposed software solutions to detect which pages to migrate to the fastest memory (e.g., DRAM). These schemes work with a horizontal implementation of hybrid memory systems when both DRAM and NVM are memory mapped and exposed to the OS. On the other hand, vertical implementation of hybrid memory systems uses the DRAM as a cache. Therefore, the DRAM is not exposed to the OS, wherein caching pages is handled using dedicated hardware, typically an extension of the memory controller as in Intel's Optane DC memory mode [8]. Schemes like the one proposed by Ramos et al. [67] rank the pages based on how frequently each page is accessed using a Multi-Queue (MQ) structure, then use the pages' ranks to decide which pages to migrate to the DRAM

and which pages to keep in the NVM. However, tracking all the pages and checking the MQ structure to promote and demote pages entails high overheads, therefore only the head of the queue is checked in each epoch.

After discussing the hybrid memory system's management schemes, we discuss some of the used schemes for page caching in HMM.

Page Caching Policy

The page caching policy is used to determine which pages should be cached in the DRAM, if used to cache the NVM pages. In this Section, we discuss two policies that we use later in our design.

First touch policy: This policy caches the pages on the first access and selects a page for eviction based on the LRU algorithm.

Multi-Queue (MQ): The MQ was originally designed to rank disk blocks, and later used by Ramos et al. [67] for page placement in hybrid memory systems. The MQ works as follows: MQ defines `M` LRU queues of block descriptors. The queues are numbered from 0 to `M-1`, with blocks at queue `M-1` are the most accessed blocks. Each descriptor contains the block's number, a reference counter, and a logical expiration time. On the first access to a block, its descriptor is placed in the tail of queue 0, and its expiration time is updated to `CurrentTime + LifeTime`. Both times are measured in the number of accesses, and the `LifeTime` represents the number of consecutive accesses to different blocks before the block is expired. Every time the block is accessed, its expiration time is reset to `CurrentTime + LifeTime`, its reference counter is incremented, and its descriptor is pushed to the tail of its current queue. After a certain number of accesses to the block's descriptor in queue `i`, it gets promoted to queue `i+1` saturating in queue `M-1`. On the other hand, blocks that have not been accessed recently get demoted. On each access, descriptors at the

heads of all queues are checked for expiration. If the descriptor is expired, it is placed in the tail of the below queue and has its life time reset, and its demotion flag is set [67]. If a descriptor receives two consecutive demotions, the descriptor is removed from the MQ structure. In order to reduce the overhead of promotion/demotion, these operations are only performed at the end of each epoch.

As it has been proven that MQ is superior to other algorithms in selecting pages to replace [67,98], it aligns with our goal, as it facilitates detecting the performance critical pages (hot pages), and selecting the non-performance critical pages for evictions. Thus, in our experiments, we use the MQ design proposed by Ramos et al. [67]. After discussing the caching policies, we now mention the currently available industrial implementations of hybrid memory systems.

Current Industrial HMM Systems

Currently, there are different types of HMM systems available in the market. For instance, JEDEC defines three different standards for HMM known as NVDIMM. NVDIMM types have different characteristics, persistency, and performance features. Moreover, Intel recently revealed details about the memory mode and application direct mode for the Optane DC.

NVDIMM-N contains a DRAM portion, a NVM portion, and a super capacitor. The system uses the DRAM in normal execution, and the NVM is only used to copy the DRAM data using the super capacitor power during crashes. [10]

NVDIMM-F module is a NVM attached to the DDR bus, the access latencies of which is relatively higher than the DRAM. Thus, a DRAM can be installed in the system and used to cache the NVDIMM-F data at the cost of data persistency [10].

NVDIMM-P is still a proposal for a DIMM that have memory mapped DRAM and NVM, wherein

the software places the data either on NVM or in DRAM, based on the size and the persistency requirements [10, 11].

Optane DC Memory Mode is an operating mode of Intel's persistent memory, which is similar to the vertical implementation of the HMM. The NVM is used as the system's main memory, and the DRAM is used to cache the NVM's content. This mode provides access to a large memory with access latencies close to DRAM, but *does not provide persistency* [8].

Optane DC Application Direct Mode is an operating mode of Intel's persistent memory, which is similar to the NVDIMM-P.

Persistent Memory Programming Model

Due to the persistency feature of NVMs, accessing an NVM memory object is like accessing a storage file. Thus, applications need a way to re-connect to previously allocated memory objects. Therefore, persistent memory regions need names and access control to be accessed. Storage Networking Industry Association (SNIA) recommended OSes to provide standard file semantics for naming, permissions, and memory mappings. Thus, Direct-Access (DAX) support for filesystems was added by several OSes [9]. DAX allows the application to directly use the persistent memory without using the system's page cache. Figure 3.1 shows how persistent memory aware filesystem works [75].

Using persistent memory (PM) objects requires the programmer to consider multiple issues to ensure the data persistency and consistency. One of these issues is `atomicity`; what kind of support is provided by the hardware, and what is left for the software to handle [75]. Intel's hardware ensures the atomicity for 8-byte writes, thus if an object is larger than 8 bytes, it is the software's responsibility to ensure the atomicity of updating the object [75]. Moreover, ensuring

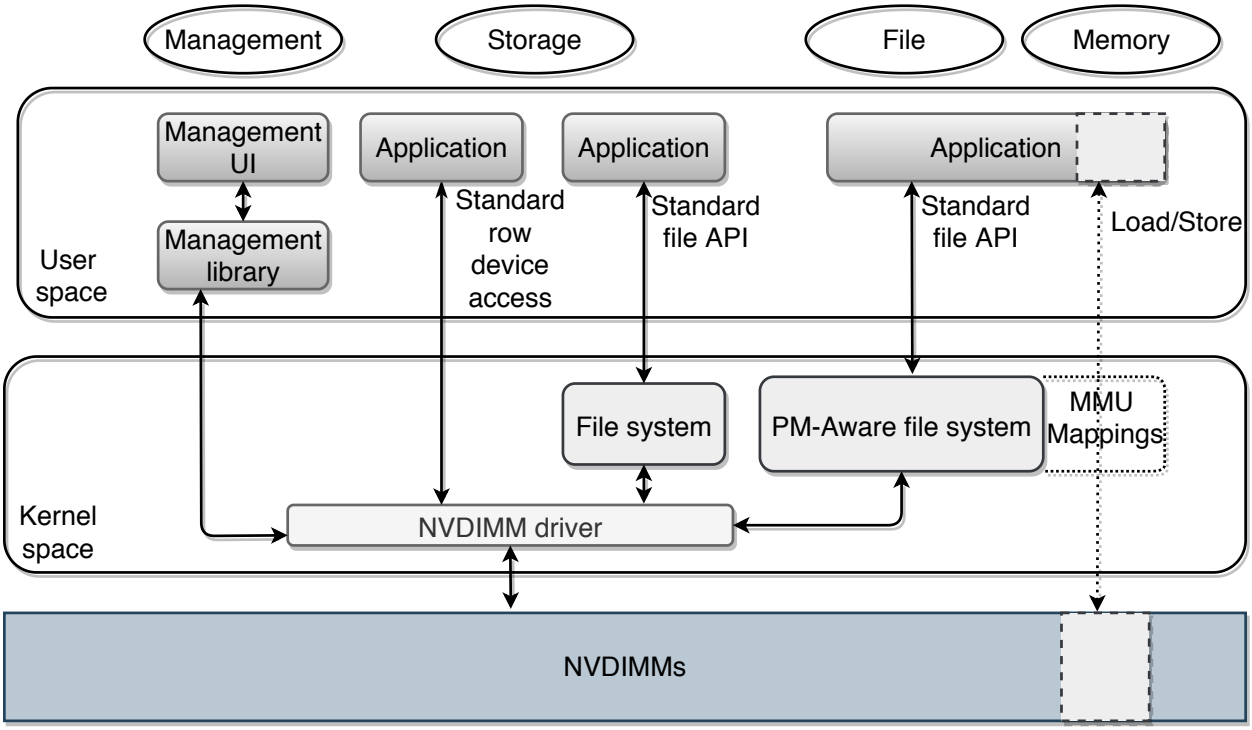

Figure 3.1: Persistent memory aware file system.

data persistency requires pushing the data all the way to the persistent domain, as most of the data updates are done in the volatile processor caches. The persistent domain starts with the `Write Pending Queue` (WPQ), which is a small buffer in the memory controller. The WPQ is supported by the Asynchronous DRAM (ADR) refresh feature. The power provided by the ADR ensures flushing the WPQ content to the NVM in case of power failure [**?**, 19, 82, 90]. Figure 3.2 shows the persistent domain in a system with persistent memory.

Listing 3.1: NVM programming example

```
1 // a, a_end in PM
2 a[0] = foo();         // store foo() in a[0]
3 msync(&(a[0]), ...); // sync to PM
4 a_end = 0;           // store 0 in a_end
```

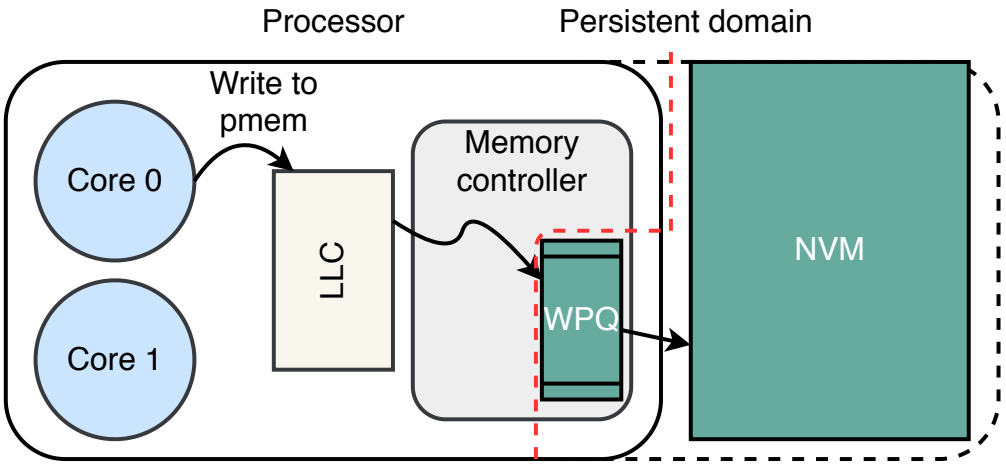

Figure 3.2: Persistent domain.

```
5  msync(&a_end, ...);  // sync to PM
6  . . .
7  n = a_end + 1;       // store a_end+1 in n
8  a[n] = foo();        // store foo() in a[n]
9  msync(&(a[n]), ...); // sync a[n]
10 a_end = n;           // store n in a_end
11 msync(&a_end, ...);  // sync to PM
```

In order to flush the data all the way to the persistent region, ensure atomicity, and ordering, a set of specific instructions need to be followed. Listing 3.1 shows a code example taken from SNIA NVM Programming Model V1.2 [12]. The code shows the persistent objects `a` and `a_end`. To ensure the persistency, atomicity, and ordering of updates to these persistent objects, `msync` operation is called each time one of these persistent objects is updated. Note that the update at line 7 was not followed by the `msync` operation as it is not updating a persistent object. The `msync` operation is used to force the updates of a memory range into the persistent domain. Moreover, it creates a barrier to guarantee that previous stores are performed before proceeding, `fsync` operation does

the same functionality for files [75].

Motivation

Having a persistent portion of the main memory enables applications with different persistency requirements. However, to ensure the data persistency, application's persistent data should be placed in the NVM portion of the memory, which hinders the performance of these applications, due to the slow access latencies of NVM. On the other hand, placing the application's data on the DRAM, will lead to better performance but fails to meet the data persistency requirement of such applications. To ensure the application's data persistency, persistent applications should follow the programming model mentioned in Section 3. As discussed earlier, available persistent memory technologies either provide small memory capacity but fast and battery-backed DRAM-based persistent region, or high-capacity NVM (no need for battery backup) but slow persistent region. The former requires system's support, bulky items, and can limit the size of persistent DRAM depending on the size of the ultra-capacitor or battery. Moreover, it requires certain DIMM changes to support backup mode. Meanwhile, the latter incurs significant performance degradation due to the slow read accesses of persistent objects. While the size of persistent application's data is unlikely to fit in the volatile caches, caching such persistent data in the much larger DRAM can provide significant read speed-ups for persistent objects. Meanwhile, expecting battery-backup, limited DRAM size, and limiting the options (e.g., vendor) of DRAM modules to be integrated in the system, are major drawbacks for the available solutions. Thus, it is important to support caching of persistent data objects in DRAM by just relying on minor changes to the processor chip.

Table 3.1 compares between the available technologies. From Table 3.1, we can observe the gap between supporting high-performance persistent memory, and high-capacity persistent memory,

Table 3.1: Technologies comparison.

Technology	**High persistent capacity**	**Persistent region performance**	**Flexibility**
NVDIMM-N	✗	✓	✗
NVDIMM-P	✓	✗	✗
Optane DC memory mode	✗(none)	✗(none)	✓
Optane DC app direct mode	✓	✗	✓
Stealth-Persist	✓	✓	✓

and hence Stealth-Persist aims to bridge this gap. Figure 3.3 shows the performance overheads for persistent applications running on Optane DC app direct mode (all persistent data is in NVM), compared to running on a system with DRAM that does not provide data persistency. From Figure 3.3, We can observe that applications running on Optane DC's app direct mode incur an average of 2.04x performance slowdown.

Design

In this section, we discuss Stealth-Persist's design in light of possible design options and their trade-offs. First, we start by discussing the design requirements, and the potential design options.

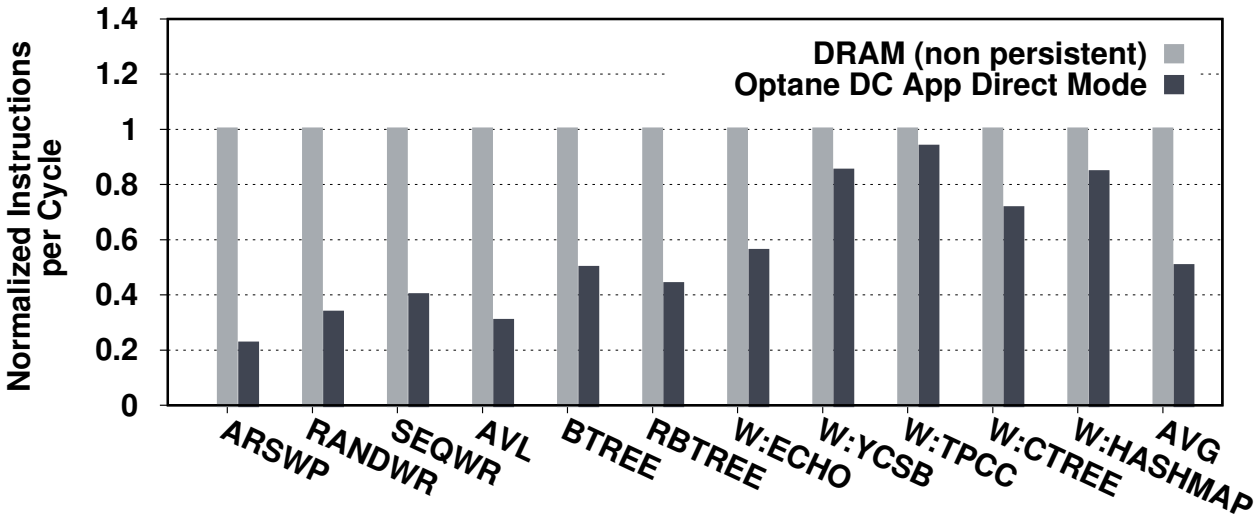

Figure 3.3: Normalized performance of persistent applications with DRAM and Optane DC app direct mode with respect to DRAM.

Design Requirements

Our design should meet the requirements necessary to allow wide adoption and high-performance, while preserving the semantics of persistent objects. In summary, the requirements are as following:

- **Flexibility:** our design should allow the integration of any DRAM module, regardless of its capacity, in a NVM-equipped system, without requiring any special battery back-ups or specific DIMM modifications.
- **Persistency:** any memory page or object that is supposed to be persistent (i.e., recoverable from crashes) should be recoverable without any extra battery backup support, regardless of where is the page located (NVM or DRAM).
- **High-Performance:** accesses to persistent pages and objects should be as fast as accesses to DRAM.

- **Transparency:** applications that leverage persistent memory for crash recovery should not need to explicitly manage caching and persisting of objects currently residing in DRAM.

To put these requirements in the context of persistent applications, we can imagine a persistent application that accesses tens of gigabytes of persistent objects. Ideally, the system should be able to have DRAM modules integrated in addition to the NVM modules. Systems' owners should have the flexibility on what capacity and vendors to choose such DRAM and NVM modules from, which provides ***flexibility***. However, updates to persistent objects should be durable and persistent across crashes, regardless of where they exist (DRAM or NVM). While updates to an object in the volatile caches are made durable through the persistency model and framework, i.e., clflushes and memory fences, there is no current support to guarantee the durability of persistent objects if they are cached in the off-chip DRAM, which brings us to the ***persistency*** requirement. Finally, the application should ideally have its persistent objects cacheable in DRAM to minimize the cost of fetching persistent objects that do not fit in the volatile processor caches, which are typically a few megabytes. The requirement to fetch off-chip persistent objects with a latency shorter than the slow NVM's latency (300ns read latency vs 70ns for DRAM) brings us to the third element of our design requirements, ***high-performance***. Thus, persistent applications should be able to cache their persistent objects, that do not fit in the internal volatile caches, in the fast off-chip DRAM, while preserving their persistence capability. Finally, all operations for caching and persisting pages of persistent objects should happen transparently to the application and software, without exposing such details to the application, which brings us to the final requirement, ***transparency***.

Design Options

We will now discuss the design options that can potentially meet our requirements.

One option is to support new instructions that do not commit until a cacheline is flushed – not only from volatile caches, but also from the off-chip DRAM, to the NVM. Such a design option can be realized by introducing new instructions to the instruction set architecture (ISA) with support from the memory controller, or by modifying the implementation of current instructions so that they flush cachelines from the internal volatile caches (e.g., `clflush`), as well as from the DRAM to the NVM. Assuming that the DRAM is operated as a hardware-managed cache for the NVM's data through the memory controller, such instructions would need to have the memory controller first check if the cacheline to be persisted is currently in the DRAM, read it, then flush it to the NVM. The main issues of this approach are: (1) it requires changes to the ISA, persistency programming libraries, and the processor core to support such new instructions. Additionally, (2) the latency to persist data will be significantly increased, especially if the flushed block is marked dirty in the DRAM. Note that even if the DRAM is caching pages instead of cachelines, it will still require similar support but with new instructions that operate at the page granularity, instead of clflush.

One another option is to leverage small fixed size backup capability (e.g., ultra-capacitor) to power flushing a specific portion of the DRAM. For instance, sufficient power to flush 8GB of DRAM, regardless of the total size of the module. The memory controller or the system's software can potentially migrate or place persistent pages in this subregion of the address space, marked as being persistent. When a power failure occurs, the memory controller (or external system circuitry) has sufficient power to flush that portion of the DRAM. While such a solution is similar in spirit to NVDIMM-N, it provides flexibility for choosing any DRAM module and capacity. However, the size of the portion has persistence support is limited to the backup capability of the system. On the other hand, such a solution requires external system support and limits the size of the persistent portion of the DRAM to the power backup capability. Again, such backup capabilities are typically costly, requires high area (bulky), and can be environmentally unfriendly.

While the first option provides *high-performance*, *persistency* and *flexibility*, it lacks the *trans-*

parency. Meanwhile, the second option has partial *flexibility* (requires system support and possibly ISA changes), partially *high-performance* (only a small portion of DRAM can be used as persistent memory), *transparency* and *persistency*. Thus, our design should provide full transparency, high-performance, persistency, and flexibility, without any additional system support or backup capabilities beyond what is already provided in most modern systems.

Stealth-Persist Design

While meeting the aforementioned design requirements, our design should also be compatible with the different ways to integrate hybrid memory systems. In particular, vertical memory mode (e.g., memory mode of Optane DC) and horizontal memory mode (e.g., app direct mode of Optane DC).

Before delving into the details of Stealth-Persist support in different integration modes, we will discuss how Stealth-Persist meets the design requirements.

To meet the flexibility requirement, Stealth-Persist is implemented to support mirroring of updates to the persistent region to NVM when cached in DRAM. Thus, it does not require any support from the system and works with any DRAM size. By mirroring updates to persistent pages cached in DRAM, the persistency requirement is met. To make our solution transparent to software, Stealth-Persist's mirroring operations occur at the memory controller and do not require any changes to the application or persistent programming library. Finally, to support high-performance access to persistent pages, our scheme serves read requests to persistent objects from the DRAM, if cached there. Figure 3.4 depicts the read and write operations in Stealth-Persist, at a high-level.

As shown in Figure 3.4, the Memory Controller Hub (MCH) handles mirroring of writes to persistent pages if cached in the DRAM, while serving read requests directly from the DRAM. By doing so, Stealth-Persist ensures the durability of writes to the NVM while allowing fast read operations

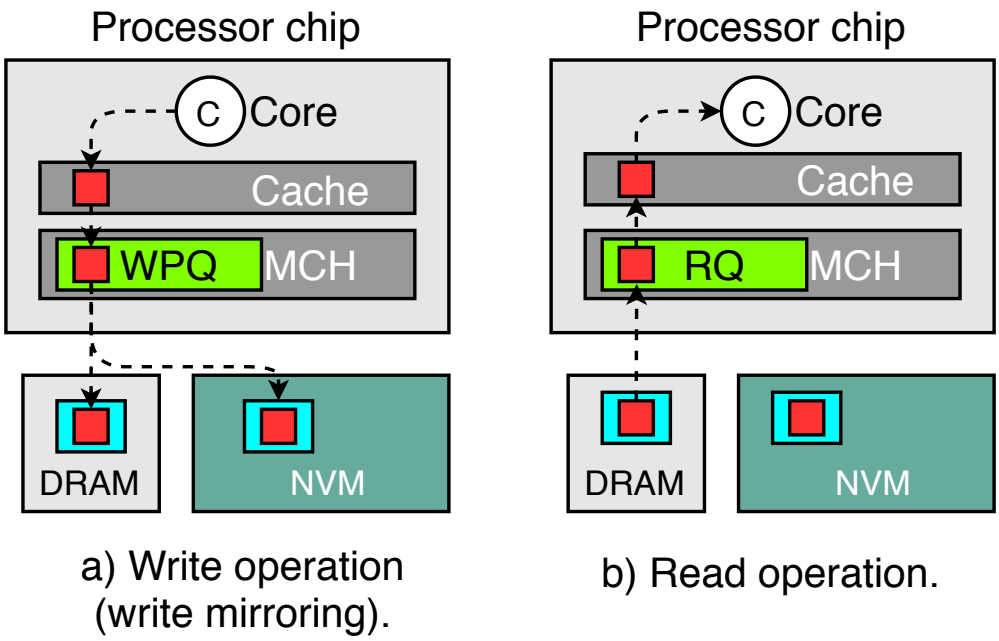

Figure 3.4: Read/Write operations in Stealth-Persist.

to such persistent objects.

While at a high-level, the design looks similar to the write-through scheme typically used in internal processor caches, many challenges and potential divergences arise when considering the context of hybrid memory systems. The first challenge is how to decide if a page should be mirrored or not. The second challenge is how to quickly identify if a page is cached in the DRAM or not, where it is cached in the DRAM, and how to guarantee that both copies are coherent during run-time. Third, since not all pages in the NVM need to be persisted, updates to pages stored in the NVM need to be selectively mirrored. Finally, Stealth-Persist needs to be adapted to work with the myriad of ways to integrate hybrid memory systems. The following parts of this section discuss these challenges and how we overcome them.

Page Mirroring

Regardless of the HMM management scheme used, horizontal (e.g., app direct mode) or vertical (memory mode), Stealth-Persist requires a part of (or the whole) DRAM to be used as a mirror region for persistent pages. In the vertical memory setup, the whole DRAM will be used as a cache for NVM, and thus, any page cached in the DRAM can be possibly mirrored to the NVM as well. Meanwhile, for the horizontal setup, since the DRAM and the NVM physical ranges are explicitly exposed to the system, we have the memory controller reserve a portion of the DRAM to be used merely as a mirror region. The remaining part of the DRAM will be exposed to the system directly as in app direct mode. Any persistent page located in the NVM can be cached in the mirror region in the DRAM regardless of the setup, i.e., the size of such region. On each memory access that targets a NVM address, we need to transparently check if the page is currently resident in the DRAM. This check is needed for both read and write operations; read operations can be served directly from the DRAM, if the accessed page is cached there, whereas write operations need to update the copy in the NVM to honor coherence between the mirrored page copies and ensure persistency. When a page is not present in the DRAM, we need to read it (or write to it) from the NVM. Since the mirror region can be thought of as a buffer/cache for persistent pages in the NVM, we need to define the insertion and evictions policies for said cache/buffer in DRAM.

For simplicity, we use a page insertion policy similar to what is used in vertical memory management schemes. By doing so, if memory mode is used, no changes are required to the management policy, except additional writes to the NVM if persistent pages are cached in the DRAM. Meanwhile, for app direct mode, the defined mirror region in the DRAM will be managed similar to the DRAM cache in memory mode, in addition to the mirroring writes to the NVM. With this in mind, we use two simple policies for page placement in the DRAM buffer: (1) first-touch policy (FTP) and (2) multi-queue (MQ) policy as proposed in prior work [67].

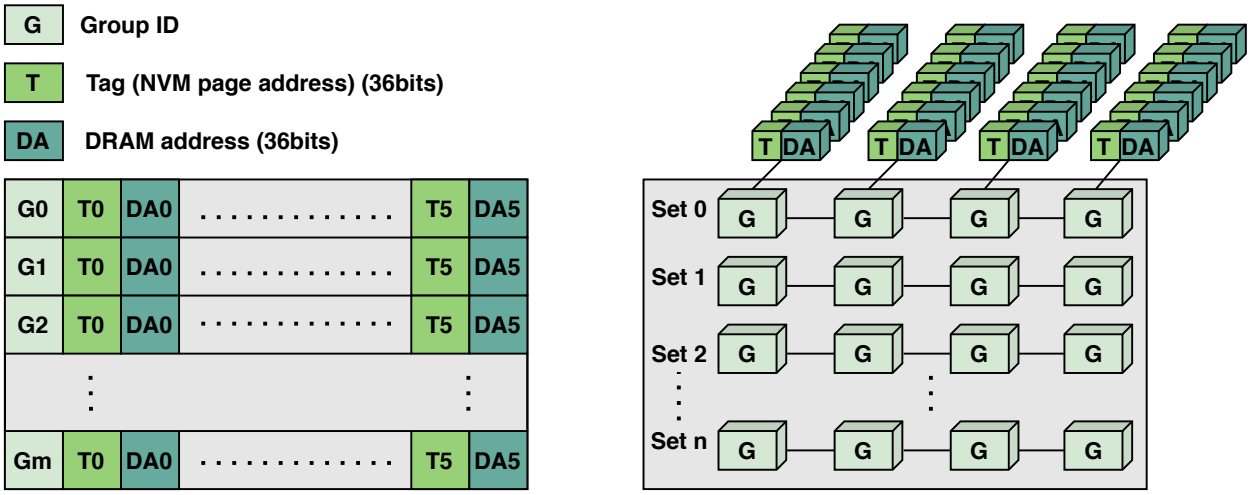

Figure 3.5: Mirroring region mapping table.

DRAM Mirror Region Lookup

To ensure Stealth-Persist can quickly check if a page is in the DRAM (mirror region) or not, Stealth-Persist keeps track of the mirror region pages using a hardware managed table. The mirror's mapping table contains the translations of the mirror's cached pages addresses, as shown in Figure 3.5. Each entry in the mapping table contains a group ID, which is calculated using a modulus function of the mirrored page address in the NVM over the number of pages in the mirror region. Additionally, each entry contains six pairs of translations that maps the 36-bit NVM's page address to the 36-bit mirror DRAM's page address. Additionally, we use 3-bits for each translation (18-bits total) as LRU bits for replacement policy in each entry, which makes a total of 450-bits for translations and the rest of the 512-bits are used for the group ID (32 bits) and padding (unused). Thus, a page can be removed from the mirror region by either the clock replacement policy or by the LRU eviction within the entry.

Note that the mirror's mapping table storage requirement is 64-bytes for every 6 pages in the

mirroring region. Therefore, we use a small cache in the memory controller to cache the mirror's mapping table entries while maintaining the table in the DRAM. Whenever a memory request to the persistent region is received, the group ID of the requested page is calculated and the mirror's mapping table cache is checked for the requested group ID, which can result in three different scenarios. ① The entry is cached and the page is cached → the request is served from associated DRAM page. ② The entry is cached and the page is not cached → the page is not mirrored and the request is served from the NVM. ③ The entry is not cached → mapping table in the DRAM must be checked to obtain the entry and its mirrored pages. Since a mapping table cache miss can lead to serving the request from the DRAM with two accesses, or from the NVM after checking the DRAM, we send the request to the DRAM and the NVM then serve the request from the DRAM, if the entry is in the table, or from the NVM if it was not.

Coherent Updates to Mirrored Pages

In Stealth-Persist, coherence between the mirror region pages and the NVM pages should be maintained. Since persistent pages are expected to be recoverable, writes to persistent pages should be durable. Therefore, writes to the mirror region should be pushed to both memories. Stealth-Persist pushes the write requests to the mirror region pages into the DRAM's volatile write buffer and to the NVM's persistent WPQ. Note that a write request is only retired once it is placed in the WPQ, which ensures the write persistency. On the other hand, mirrored pages that belong to non-persistent region do not require data coherence nor recoverability, which is why Stealth-Persist implements selective mirroring.

Stealth-Persist does not have any impact on coherence implementation. If the DRAM and the NVM modules are on the same socket, which is the configuration supported for Intel's DC PMM, coherence between the NVM and DRAM copy is managed by the MC through mirroring, whereas

coherence with internal processor caches is handled in conventional systems. However, if we deviate from the current standard of having the NVM and the DRAM on the same socket, i.e., each is on a different socket, then we can designate the memory controller near the NVM as the master, and thus it will be responsible to handle mirroring, remapping, etc., and accordingly forward any requests that hit in the mirror table cache to the memory controller in the socket has the DRAM module.

Selective Mirroring

Stealth-Persist implements selective mirroring techniques to reduce the number of writes to the NVM, which can be done by committing the writes directed to the non-persistent region to its DRAM mirrored version only. Stealth-Persist implements selective mirroring in the vertical HMM implementation just as in the Optane DC's memory mode, and in the horizontal HMM implementation as in the Optane DC's app direct mode. In both cases, Stealth-Persist requires the address range of the persistent memory region, which can be passed to Stealth-Persist by the kernel during system bring-up – for example, the Linux command memmap=2G!8G could be used to reserve a 2GB persistent region starting at address 8G. Note that forwarding the writes of the pages in the non-persistent region to its mirrored version only, violates the coherency of these pages. However, since the pages are in the non-persistent region, and these applications are not expected to be recoverable, the writes can be committed to the mirrored page only, while the whole page will have to be written back to the NVM if the page gets evicted.

Overall

The overall Stealth-Persist design is shown in Figure 3.6. For every last level cache (LLC) miss, first the memory controller checks if the request is to the persistent region or not ①. If the request

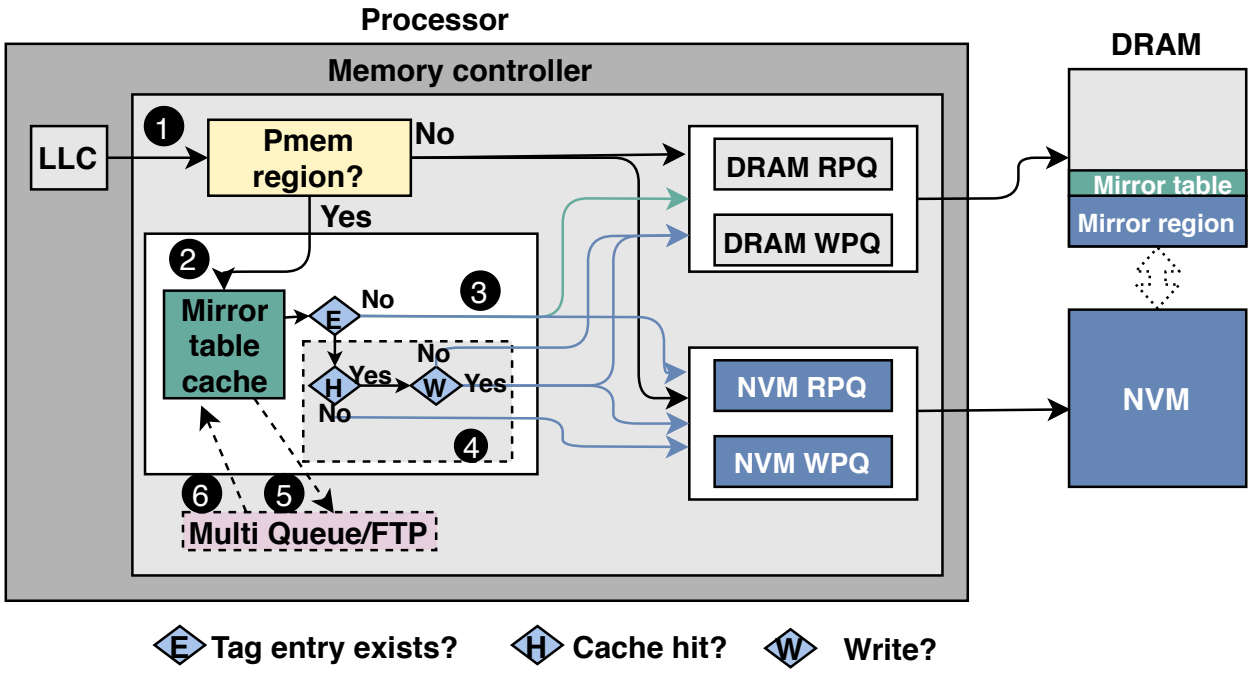

Figure 3.6: Stealth-Persist overall design.

is to the persistent region, the mirror table cache is queried for the current status of the NVM page ②. As discussed in Section 3, the mirror table cache verifies the mirroring status of the NVM's page by either looking into the already cached mirror table entries, or by fetching the entries from the mirror table stored in the DRAM, and replacing a group ID and the respective mapping table entry using a LRU policy ③. If the page is mirrored, read requests are forwarded to the DRAM while write requests are forwarded to both the DRAM, to update the mirroring region, and to the NVM, to persist the data ④. In the case of a read, the persistent memory access is forwarded to the multi-queue or FTP unit ⑤. This unit decides if a page should be mirrored and if so, the mirror table cache is triggered to replace one of the mappings using the LRU policy ⑥.

Stealth-Persist versus NVM libraries

Several studies proposed the use of NVM libraries to address atomicity, crash consistency, and performance issues when NVMs are used as a main memory. NVM libraries focus on moving writes out of the critical path to improve the performance, but do not reduce read latency. In contrast, Stealth-Persist improves the performance by reducing the latency of the basic memory read operations, which is still required with NVM libraries. Some schemes focus on fault tolerance (e.g., Pangolin [94]), performance and strong consistency (e.g., NOVA [87]), programming effort reduction and performance (e.g., Pronto [58]). While such schemes improve the system's performance by moving the writes overhead out of the application's critical path, or by buffering some of the updates in the DRAM, the writes to the NVM are inevitable if persistency is required. In contrast, Stealth-Persist propagates the writes to the NVM if they are directed towards a persistent region within the NVM, and buffers the writes to the non-persistent region in their DRAM cached pages. Additionally, Stealth-Persist operates in a different layer than the proposed NVM libraries, which makes Stealth-Persist orthogonal to such schemes. As a matter of fact, Stealth-Persist can be used concurrently with the mentioned schemes to improve the performance even further.

In a different direction, Hagmann [36] proposed a scheme that maintains a log to recover the filesystem in disks. Petal [51] enables the clients to access distributed disks by creating virtual disks, which improves the system's performance and increases the throughput. To provide the recoverability, Petal uses write-ahead-logging. Condit et al. [29] proposed a scheme that enables crash consistency for persistent memories using shadow paging, in which the writes are atomically committed in-place or using localized copy-on-write. BTRFS [69] for the Linux filesystem uses B-tree data structure, and uses copy-on-write as the update scheme. Rosenblum and Ousterhot [74] proposed a log structured filesystem that performs all the writes to the disk in a sequential manner, and maintains indexing information for faster data retrieval. Seltzer et al. [78] proposed

a log-structured filesystem that has improved write performance, less recovery time, and enables embedded transactions and versioning. Such schemes were proposed to ensure atomicity, recoverability, and improve the performance in disks. However, the proposed schemes do not improve the performance of read operations while ensuring persistency. Thus, they are orthogonal to Stealth-Persist.

Methodology

We modeled Stealth-Persist in the Structural Simulation Toolkit (SST) simulator [70]. SST is a cycle-level event-based simulator with modular designs for different hardware components. SST is widely used in the industry and academia [44,47,48]. We implemented a hybrid memory controller component to handle both DRAM and NVM. Stealth-Persist required components, Mirroring-Table and the MQ, are modeled in a hybrid memory controller module to perform all the relevant tasks. The configuration of the simulated system is shown in Table 5.2. The simulated system contains 4 out-of-order cores with each core executing 2 instructions per cycle. The frequency of the cores is 2GHz. Three levels of caches, L1, L2, and L3 (inclusive) are simulated with sizes 32KB, 256KB, and 1MB respectively. The DRAM capacity is 1GB and the NVM capacity is 4GB[1]. NVM read and write latencies are 150ns and 500ns [17]

Workloads:

To evaluate our proposed scheme, we ran 11 persistent applications. As shown in Table 5.4, six of the benchmarks were developed in-house, all of which are designed to stress memory usage and

[1]Note that the selected sizes of DRAM and NVM are chosen due to the limitation of simulation speed, however, the most important parameters are the mirroring region size (32MB) and the average footprint of the applications (256MB). Since all the data of persistent applications will reside in NVM and can be cached persistently in the mirroring region, we focus on the ratio of application's footprint to the mirroring region (8:1 ratio), which we vary later in the work.

Table 3.2: Configuration of the simulated system.

Processing Element	
Processor	4 Cores, X86-64, Out-of-Order, 2.00GHz, 2 issues/cycles, 32 max. outstanding requests.
L1 Cache	Private, 4 Cycles, 32KB,8-Way
L2 Cache	Private, 6 Cycles, 256KB, 8-Way
L3 Cache	Shared, 12 Cycles, 1MB/core, 16-Way
Cacheline Size	64Byte
Hybrid Main Memory	
DRAM	Size: 1GB, RCD=RP=14, CL=14 CL_WR=12
NVM	Size: 4GB, Read latency 150ns, Write latency 500ns
DRAM Mirror	
Size	32MB
MQ mirroring threshold level	4
Epoch interval	10000 reads
Mirroring Table cache	size: 128 entries (groups), associativity: 4, latency: 1 cycle

were used in previous work [56]. The functionality of each of these applications is described as follows.

❶ ARSWP: This benchmark randomly chooses two keys from the database and swaps them.

❷ RANDWR: Random keys are chosen and the database entry with the chosen key is updated with a random value.

❸ SEQWR: This is similar to RANDWR but the keys are chosen sequentially starting from the 1st element of the database.

❹ AVL: The database is mapped to an AVL tree and a randomly generated key is searched in the mapped database. If the key is not found an insertion operation is triggered.

❺ BTREE: This benchmark maps the database to a B-tree and similar to AVL, a random key is searched, if not found the key is inserted with dummy data.

❻ RBTREE: Similar to AVL and BTREE benchmarks, RBTREE benchmark maps the database to a RB-tree and a random key is searched.

We also ran five benchmarks from the WHISPER benchmark suite [60] (preceded by W: in Table 5.4) developed by the University of Wisconsin-Madison in collaboration with HP Labs. The TPCC benchmark measures the performance of online transaction processing systems (OLTP) based on a complex database and various database transactions that are executed on it. The Yahoo Cloud Serving Benchmark (YCSB) is a programming suite to evaluate database management systems. W:TPCC and W:YCSB benchmarks are variants of the Whisper benchmark suite that are modeled after N-Store [16], which is a remote data base management system for persistent memory. W:CTREE and W:HASHMAP benchmarks were developed using the NVML [83] library which performs insert, delete and get operations to the persistent memory regions. W:ECHO is a scalable key-value store for persistent memory regions. Map_get functionality is evaluated for W:CTREE and W:HASHMAP benchmarks.

The key size of all these benchmarks is 512 bits and the database size is 1GB. Before evaluating these benchmarks, first the database is filled with random keys. Misses per kilo instructions (MPKI) for these benchmarks are shown in Table 5.4. Each benchmark is evaluated for 500M instructions.

DRAM Mirror Configuration:

To mirror the NVM's data, a 32MB of the DRAM is used. However, we vary the size of the mirroring region from 2MB to 1GB (entire DRAM is used as mirroring region) as discussed in Section 5. Mirroring is done at page granularity. In MQ mechanism a page is mirrored only when it reaches MQ level 4, i.e., when a page is read 16 times. The epoch interval is set to 10000 read operations. Although we evaluated Stealth-Persist approach with the above-mentioned configurations[2], we performed sensitivity analysis by varying the DRAM mirror size, and threshold

[2]We used CLWB to persist the data and keep the data in the processor caches.

Table 3.3: Benchmarks description.

Benchmark	Description	MPKI
ARSWP	Swap random elements of an array	31.11
RANDRW	Random updates to persistent memory	32.43
SEQRW	Sequential updates to persistent memory	6.18
AVL	Insert and look up random elements in avl tree	30.38
BTREE	Insert and look up random elements in b-tree	21.01
RBTREE	Insert and look up random elements in red-black tree	56.11
W:YCSB	N-Store variant to evaluates database management systems	3.88
W:TPCC	N-Store variant to measures the performance of online transactions	3.97
W:CTREE	NVML variant of crit-bit tree	1.75
W:HASHMAP	Hashmap implemented with NVML [83] library	0.84
W:ECHO	Scalable key-value store for persistent memory	9.54

level. Mirror table cache size maintained by the memory controller is 128 groups with each group having 6 mappings. Mirror table cache lookup latency is 1ns.

Evaluation

In this Section, we discuss the results of Stealth-Persist against a system using the NVM directly for persistency. We further show sensitivity analysis by varying different parameters that impact the performance.

Impact of Stealth-Persist on Performance

Figure 3.7 shows the performance improvement with Stealth-Persist methods. The baseline scheme is the Optane DC app direct mode scheme wherein all the persistent memory requests are stored to the persistent memory (NVM) only. This is the typical way of achieving data persistency for

persistent applications for such systems. On average, the performance improves by 30.9% and 42.02% with Stealth-Persist MQ and FTP approaches. The application's performance improvement is a function of the mirroring region's hit rate as discussed in Section 3. Improvement with Stealth-Persist FTP is higher than Stealth-Persist MQ method since every page that is read is mirrored in the DRAM, which leads to a huge number of pages copied from NVM to DRAM. On average, we observe Stealth-Persist FTP mirrors 542.96x more pages than Stealth-Persist MQ approach, which significantly increases the memory bus traffic and energy use. For sequential memory access benchmarks like SEQWR and W:ECHO, the improvement with Stealth-Persist FTP is substantial – 2.34x and 2.2x respectively. Since such benchmarks access the memory sequentially, the spatial locality for these benchmarks is high. Hence, when a page is read, it is mirrored immediately in Stealth-Persist FTP and is accessed for the contiguous memory accesses. On the other hand, Stealth-Persist MQ approach, first, the page should reach a threshold to be mirrored. For AVL and RBTREE workloads, Stealth-Persist MQ approach outperform Stealth-Persist FTP because Stealth-Persist FTP replaces the pages in the mirroring region very frequently, which leads to evicting hot pages from the mirroring region. On the other hand, Stealth-Persist MQ approach tends to keep hot pages in the mirroring region.

For the ARSWP workload, the performance of Stealth-Persist scheme barely changes compared to Optane DC app direct mode and, from Figure 3.3, it suffers significantly compared to a system using DRAM as main memory – it is 4.39x slower. However, the ARSWP application memory accesses are very sparse, and thus the reuse distance of the pages are high, which leads to evicting those pages in Stealth-Persist FTP approach before they are reused. Additionally, the pages of the ARSWP application do not reach the mirroring limit for Stealth-Persist MQ approach. Hence, the performance degrades by 3% in MQ approach due to checking the mirror region while having only 0.02% hit rate. On the other hand, Stealth-Persist FTP performance improves by 1.6% for ARSWP benchmark due to having 3% hit rate. However, the performance of ARSWP improves when the

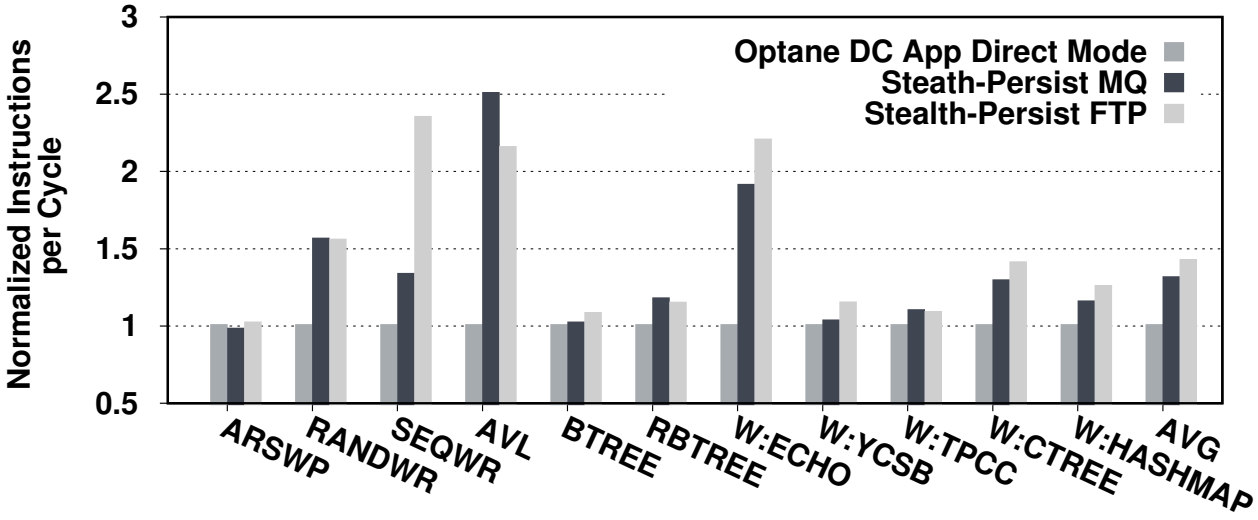

Figure 3.7: Normalized performance improvement of Stealth-Persist methods compared to Optane DC app direct mode.

mirroring region size is increased, as shown in Section 3.

DRAM Mirror Hit Rate

Figure 3.8 shows the percentage of reads served by the DRAM mirroring region. We note that applications with sequential memory accesses show the best performance improvement – FTP is showing a very high hit rate for these applications. On the other hand, applications with random stride accesses and ones with hot pages, show the highest hit rates in Stealth-Persist MQ approach. As Figure 3.8 illustrates, the mirrored pages serve an average of 57.81% of the overall memory reads in Stealth-Persist FTP approach. For Stealth-Persist MQ, it serves an average of 24.78% of the overall reads with a reasonable number of page mirrors compared to Stealth-Persist FTP. As shown in Figure 3.8, memory bounded applications with the highest hit rates show the highest performance improvement. In Stealth-Persist FTP, the mirroring hit rate for WHISPER benchmarks, like CTREE and HASHMAP is high, but the performance improvement is not as much as for SE-

QWR and ECHO benchmarks. This is because CTREE and HASHMAP applications are not as memory intensive as EPOCH and SEQWR, which is correlated with the MPKI for CTREE and HASHMAP, as shown in Table 5.4 – CTREE has an MPKI of 1.75 and HASHMAP has an MPKI of 0.84.

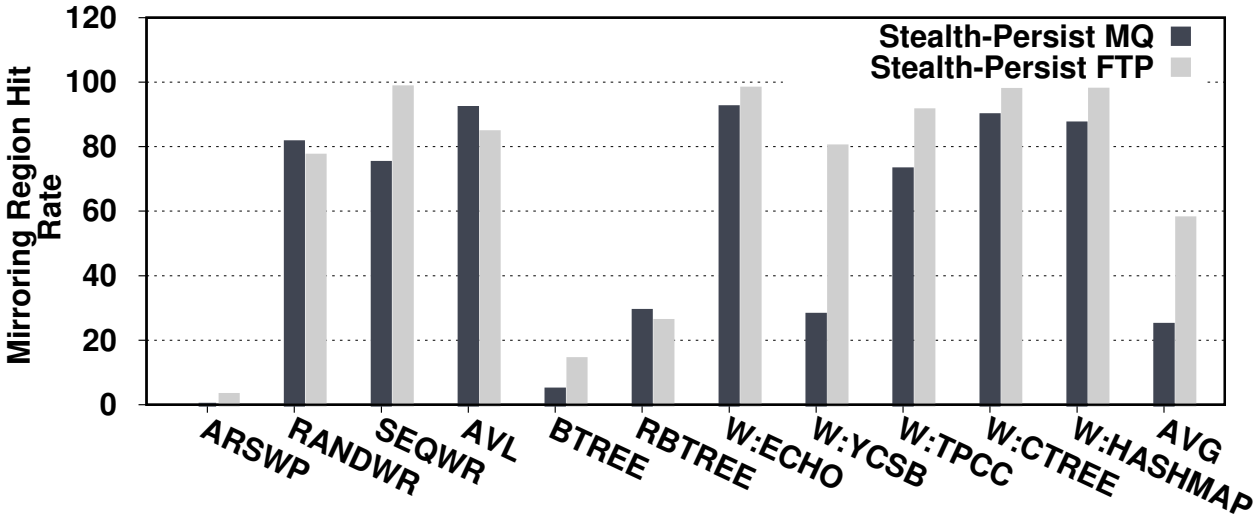

Figure 3.8: Percentage of requests served by the mirroring region.

Impact of Stealth-Persist on NVM Reads

In this section, we show the reduction in the number of reads sent to the NVM using Stealth-Persist approaches. When the mirroring region hit rate is high, most of the reads are served by the mirroring region, which reduces the number of reads sent to the NVM. Figure 3.9 shows that, on average, the number of NVM reads are reduced by 88.28% and 73.28% with Stealth-Persist FTP and MQ approaches, with respect to Optane DC app direct mode (100%). For the SEQWR and W:ECHO benchmarks, which show the highest performance improvement with Stealth-Persist FTP, NVM reads are significantly reduced by 98.42% and 98.02%, respectively.

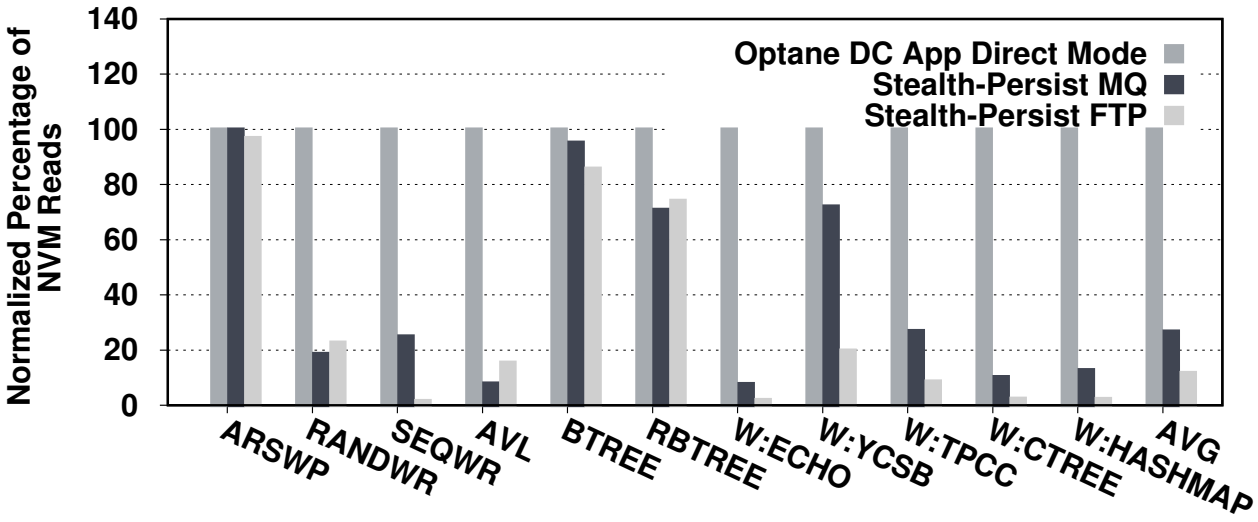

Figure 3.9: Percentage of reads served by NVM with Stealth-Persist methods compared to Optane DC app direct mode.

Impact of Stealth-Persist on NVM Writes

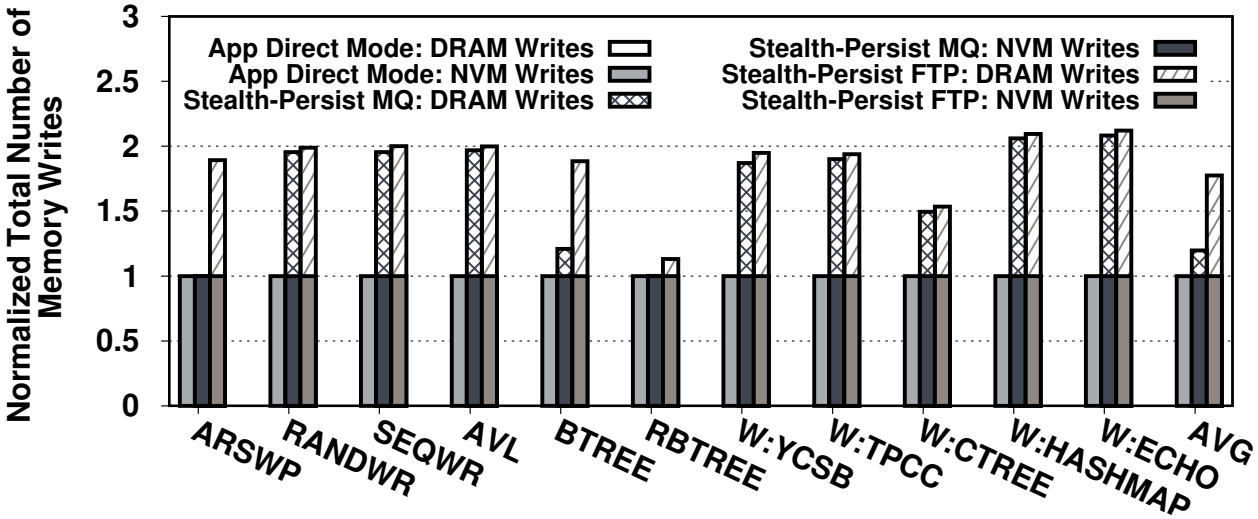

Figure 3.10: Number of writes to DRAM and NVM with Stealth-Persist methods compared to Optane DC app direct mode normalized to NVM writes.

As Figure 3.10 shows, Stealth-Persist schemes do not have any impact on the number of writes

to the NVM. However, Stealth-Persist sends the writes of the mirrored pages to the DRAM as well. Therefore, Stealth-Persist does not affect the NVM's write endurance nor increase the energy consumption, which might be caused by increasing the NVM writes.

Sensitivity analysis

Although Stealth-Persist FTP and MQ improve the performance by 42.02% and 30.9% on average compared to the baseline (Optane DC app direct mode), there is still a room for improvement since the mirroring region hit rate is 57.81% and 24.78%, on average. Misses can happen for many reasons, but are mainly affected by the mirroring region size and mirroring threshold in Stealth-Persist design. However, increasing the mirroring region size will increase the hardware complexity (Mirroring-Table size) while reducing the mirroring threshold may result in early replacement of required pages, which may degrade the overall performance. To fully analyze the effects of the mirroring region size and the mirroring threshold, we vary the mirroring region size and the mirroring threshold in this section. Also, we show the performance improvement on fast and slow NVMs. The average of all the workloads is shown in the sensitivity results.

Impact of Mirroring Region on Performance

The number of persistent pages that can be mirrored in the DRAM is dependent on the percentage of the DRAM memory reserved for mirroring. To avoid significant memory overhead, Stealth-Persist reserves only 32MB of the DRAM, which is 3.125% of the DRAM in the simulated system, for mirroring of persistent memory pages. However, as discussed previously, the more pages that can be mirrored, the greater the upper bound on system performance when using Stealth-Persist. Hence we varied the mirroring region size from 2MB to 1GB to evaluate performance improvements with Stealth-Persist. Note that when the mirroring region size is 1GB, the entire

DRAM is reserved to cache mirroring pages.

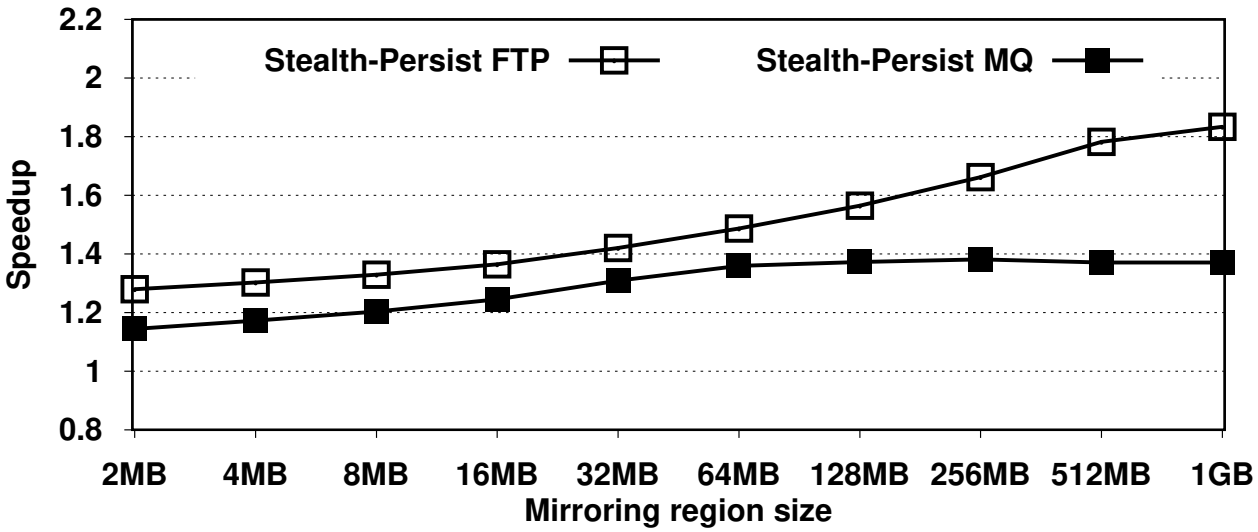

Figure 3.11: Performance improvement with different mirroring region sizes.

Figure 3.11 shows that increasing the mirroring region size improves the performance of both FTP and MQ. As the mirroring region size increases from 2MB to 1GB, the performance improvement increases from 1.28x to 1.83x with Stealth-Persist FTP and increases from 1.14x to 1.38x with Stealth-Persist MQ. The improvement is saturated after 64MB mirroring region size with Stealth-Persist MQ, since MQ is a confirmation based approach wherein a NVM page is mirrored only if it is accessed for more than the threshold number of times(4). Hence, even though the mirroring region size is increased, the number of pages to mirror is bounded by the threshold and hence performance improvement is saturated. When mirroring region size is 64MB, the performance improvement with Stealth-Persist FTP is 1.48x and 1.35x with Stealth-Persist MQ. Also, as asserted, ARSWP benchmark which is not showing performance improvement with 32MB mirroring size, achieves an improvement of 1.06x, 1.22x, 1.75x, 2.65x, and 3.22x when the mirroring region size is 64MB, 128MB, 256MB, 512MB, and 1GB with Stealth-Persist FTP, respectively. However, with Stealth-Persist MQ we observe no improvement since the pages of the ARSWP application do not reach the mirroring threshold.

Mirroring Threshold Level Impact on Performance

In Figure 3.12, we show the results when varying the mirroring threshold queue level. When the threshold level is decreased, the performance improvement with Stealth-Persist MQ approach is increased. We observe a performance improvement of 1.46x when the threshold level is set to 1 and, with a threshold level of 4, the performance improvement is 1.3x. Stealth-Persist behaves aggressively when the threshold level is reduced since more pages are identified as mirroring candidates. That is, when the threshold level is 1, a page is identified as a mirroring candidate if the application reads the page at least 2 times. But, when the threshold level is 4, a page is mirrored only if it is read a minimum of 16 times. Hence, the performance improvement achieved by reducing the threshold level is at the cost of increasing the number of pages to mirror.

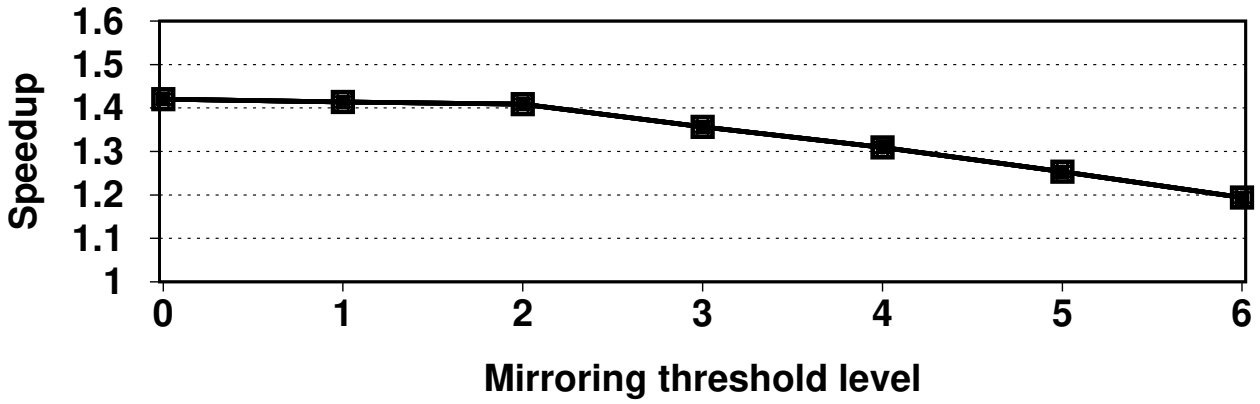

Figure 3.12: Performance improvement by Stealth-Persist MQ by varying the mirroring threshold level.

On the other hand, increasing the threshold level can hurt the performance improvement due to two reasons. 1) A page is mirrored after reaching the threshold level, as the queue level increases, and the application has to access the page more frequently to be identified as a mirroring candidate. In general, the percentage of these pages is small and they are often cached in the processor. 2) The hotness of the page is lost after reaching the threshold level. For instance, if the threshold level

is set to 6, a page has to be accessed for a minimum of 64 times to be mirrored. However, after accessing the page for 64 times, the application may no longer need access to this page, negating the impact of mirroring.

Impact of NVM Read/Write Latency on Performance

Although the NVM's read latency is comparable to the DRAM's read latency, it is still slower than the read latency of the DRAM. The write latency of the NVM suffers significantly compared to the DRAM. NVM's read/write latencies are critical while mirroring pages from the NVM to the DRAM. Hence we study the impact of Stealth-Persist for slow and fast NVM's read/write latencies. We varied the NVM's read and write latencies as shown in Figure 3.13. Figure 3.13 categorizes the NVM into 4 types - moderate; read and write latencies are 150ns and 500ns, slow: read and write latencies are 300ns and 700ns, very slow: read and write latencies are 500ns and 900ns and ultra slow: read and write latencies are 750 and 1000ns. As the NVM's read/write latencies increase the performance improvement also increases with Stealth-Persist. With ultra-slow NVM, Stealth-Persist improves the performance by 1.87x and 1.54x with FTP and MQ respectively.

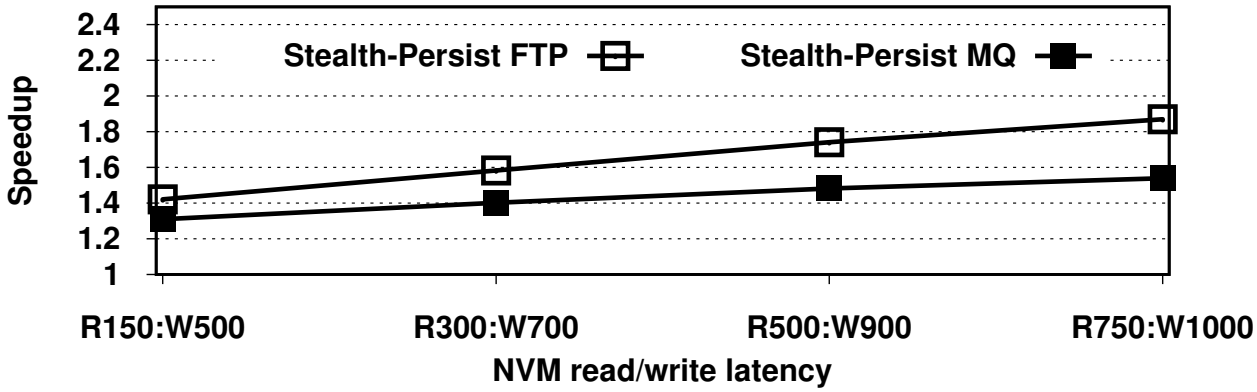

Figure 3.13: Performance improvement with Stealth-Persist for different NVM's read/write latencies compared to Optane DC app direct mode respectively.

Related work

Hybrid Memories: Previously a lot of work has been explored to improve the performance of hybrid memory systems. For instance, Ramos et al. [67] proposed a scheme for page placement in hybrid memory systems. The proposed scheme uses a multi-queue to rank the pages and only migrates the performance critical pages to the DRAM. However, the scheme does not ensure data persistency and is only focused on placing the performance critical pages in the DRAM. The authors of HetroOs [41] proposed an application transparent scheme that exploits the application's memory usage information, provided by the operating system, to decide where to place the data in heterogeneous memory systems. However, the motivation in HetroOs is purely for system performance and does not provide persistency guarantees. Therefore, applications with persistency requirements would still have to suffer the high NVM latency. The authors of Nimble [89] proposed a scheme that reuses the operating system page tracking structures to tier pages between memories. Additionally, Nimble provides several optimizations such as transparent huge page migration and multi-threaded page migration, which leads to 40% performance improvement compared to native Linux system. However, Nimble improves the page migration between memories and does not ensure the data persistency. Agarwal et al. [14] proposed a page placement scheme for GPUs in hybrid memory systems. However, the proposed scheme migrates pages between memories based on the application bandwidth requirements, which does not consider the data persistency. Yoon et al. [92] devised a policy that enables DRAM to cache pages with high frequency of row buffer misses in the NVM memory. CAMEO [25], PoM [79], Mempod [63] and BATMAN [26] discussed the the possible relaxations to maximize overall memory bandwidth. The proposed techniques rely on the compiler support or Linux kernel to detect pages of interest. Migrating remote pages to the local memory in disaggregated memory systems is explored by Lim et al. [54] and Kommareddy et al. [47]

NVM Data Persistency: Ensuring the persistency, performance, and crash consistency of NVM resident data has been under the spotlight recently. For instance, Janus [56] improves the persistent applications write latency by decomposing the back-end memory operations into smaller sup-operations, then overlapping the sup-operations. Intel's PMDK [13], REWIND [23], NV-Heaps [27] and LSNVMM [39] provide software based high level interface for the programmers to ensure the data persistency and provide crash consistency support. Hardware based approaches provide consistency using transactions and low-level primitives [40, 46, 61, 96]. The proposed scheme, Stealth-Persist optimizes persistent workloads read operations in hybrid memories and is orthogonal with the previous approaches. For instance, write performance can be improved with Janus while Stealth-Persist optimizes reads.

Conclusion

Improving the performance of persistent applications in hybrid memory systems requires caching the NVM resident data in the DRAM. However, caching the persistent applications data in the DRAM nullifies the persistency of those cached pages. Ensuring the persistency of DRAM cached pages can be achieved by power-backing the DRAM. However, using batteries to power-back the DRAM is expensive, unreliable, incompatible with legacy systems, and is not environmentally friendly. Therefore, we propose Stealth-Persist, a novel memory controller design that allows caching the NVM resident pages in the DRAM while ensuring the pages persistency. By serving NVM requests from DRAM, Stealth-Persist exploits bank level parallelism which reduces the memory contention and brings in additional performance gains. Stealth-Persist improves the system performance of persistent applications in hybrid memory systems by 42.02% on average with Stealth-Persist FTP. However, Stealth-Persist FTP requires significant number of pages to be copied from the NVM to DRAM. With Stealth-Persist MQ approach, we show a performance im-

provements of 30.09% with reasonable page mirrors. Stealth-Persist achieves this improvement at the cost of a small hardware managed table, a small cache in the memory controller, and by utilizing the WPQ.

CHAPTER 4: SECURITY METADATA CACHING TECHNIQUES IN FAM ARCHITECTURE

Background

Threat Model

In this work, we assume a similar threat model as in state-of-the-art work in secure memory architecture [15, 19, 20, 55, 76, 77, 90, 91, 99]. The trust base is limited to the processor and its internal structures. We assume an attacker who can snoop the local memory bus and the global memory bus, scan the memories content, tamper with memories content, and replay old packets. Differential power attacks, electromagnetic inference attacks, and attacks targeting the processor speculative execution such as Spectre and Meltdown are beyond the scope of this work.

Emerging Non-Volatile Memories (NVMs)

Emerging Non-Volatile Memories (NVMs) are expected to replace the DRAM as main memories [15, 19, 76, 77, 81, 90, 91, 99]. Emerging NVMs combine the features of main memory and storage, as they feature byte addressability, access latencies comparable to DRAM, near-zero idle power consumption, high density, and the ability to retain data during power failure episodes. Data persistency of NVMs is probably the most promising feature as it enables persistent applications such checkpointing and file systems. However, the persistency feature facilitates the data remanance attacks [90]. Therefore, NVMs are typically shipped with confidentiality protection and integrity verification features [**?**]. However, NVMs suffer from power consuming writes, and limited write endurance. In a matter of fact, the most promising NVM technology, Phase-Changed

Memory (PCM), can only endure tens of millions of writes [18]. Adding encryption to NVMs exacerbates the write endurance problem, due to the encryption diffusion property. Moreover, updating a data cacheline in a Merkle-Tree integrity protected system can lead to tens of Merkle-Tree updates. Thus, NVM friendly encryption and integrity verification algorithms are being explored by the research community.

Counter Mode Encryption

Split counter-mode encryption is used in state-of-the-art implementations of secure memory architecture [19, 55, 76, 90, 91, 99]. Counter mode encryption thwarts dictionary based attacks, bus snooping attacks, and known-plaintext attacks. Additionally, Counter mode encryption does not propagate errors as the input of a stage does not depend on the output of previous stages. Moreover, Counter-Mode encryption overlaps the One-Time-Pad (OTP) generation with memory read latency, thus hides the decryption latency except for the XOR operation. Figure 4.1 shows the split counter-mode encryption scheme. Split counter-mode encryption assigns a minor counter (7-bit) for each data cacheline, and a major counter (64-bit) for each page. An Initialization Vector (IV) composed of the page ID, page offset, minor counter, major counter, and padding. A secure processor key is used to generate the OTP by encrypting the IV using AES encryption engine. Then, the cacheline is XORed with the OTP to do the encryption/decryption. To ensure the security of counter mode encryption, re-using encryption counters is prohibited as it facilitates known-plaintext attacks [19, 90, 99].

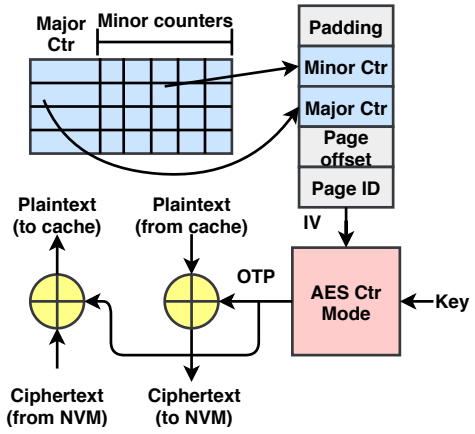

Figure 4.1: Split-Counter Mode Encryption

Integrity Trees

Bonsai Merkle-Tree (BMT)

While the integrity of the data can be easily verified using a keyed Message Authentication Code (HMAC) values calculated over the data and the encryption counters [34,71], it would be sufficient to protect the encryption counters using a hash tree with its' root kept secure in the processor. The BMT is a tree of hashes built on top of the encryption counters to ensure the integrity of the encryption counters. The BMT calculates the hash of encryption counters as shown in Fig.4.2 to create the first level of the tree. Then, it calculates the hashes of first level nodes to generate the second level and so on. The processes of hashing is continued recursively until a single node is calculated, which is referred to as the root. The BMT calculates the hash of 64 encryption counters to generate the first level, and hashes each 8 nodes (arity of eight) to form upper levels. Figure 4.2 shows a BMT with arity of two.

Whenever an encryption counter is fetched from the memory, its integrity needs to be verified by calculating the hashes until a root is generated. If the calculated root matches the processor stored

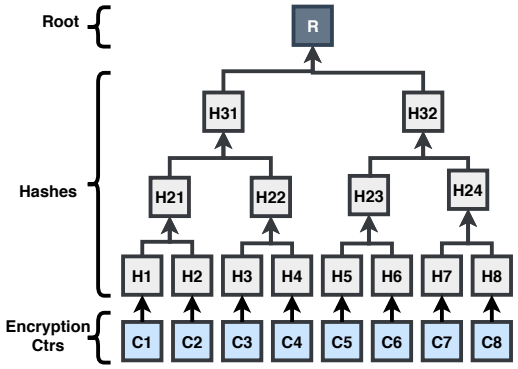

Figure 4.2: Bonsai Merkle-Tree

root, the encryption counter integrity is verified. Similarly, when a dirty encryption counter is written to the memory, the whole BMT branch needs to be updated. A faster way to verify the counter integrity can be achieved by stopping the verification process with the first parent cache hit, as the cached nodes' integrity was verified when it was brought to the processor cache.

Tree of Counters (ToC)

The ToC shown in Fig.4.3 is a parallelizable form of the MT. The ToC uses 56-bit counter for each data cacheline, and packs each eight counters (arity of eight) along with a 56-bit MAC value, and an unused eight bits in a single cacheline [34, 99]. The node's MAC value is calculated over the node counters along with a counter from the parent node. The integrity verification in the ToC is similar to the BMT, but the update process is different. Whenever an encryption counter is updated, the corresponding counter in the parent node is incremented and the MAC value is updated. Since the upper nodes' update does not depend on the update of the child nodes, the update process can be done in parallel [34, 99].

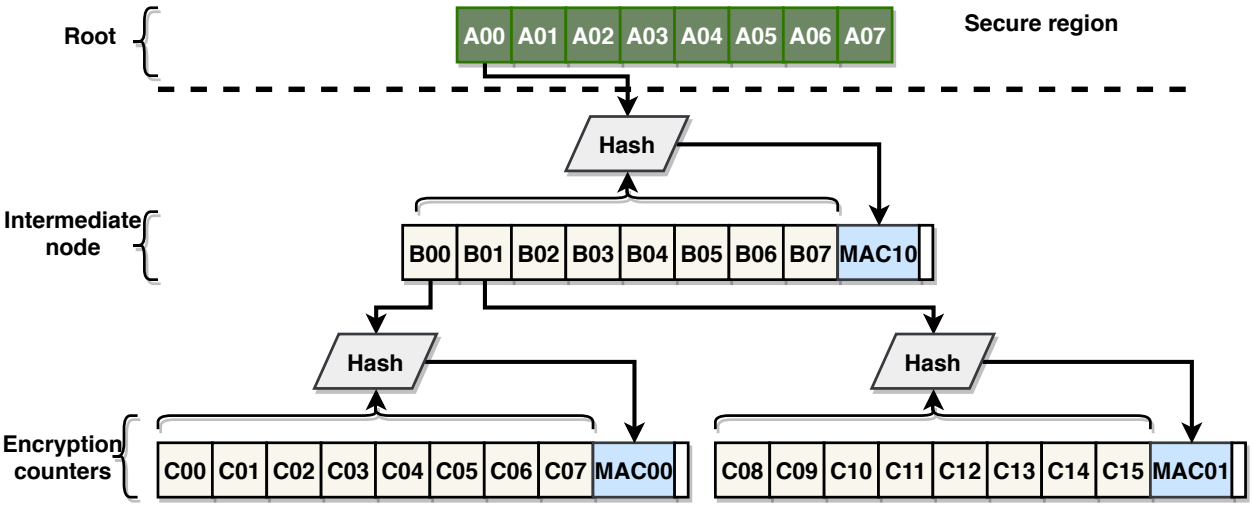

Figure 4.3: Tree of Counters.

Integrity Tree Update Schemes

Both BMT and ToC can be either eagerly or lazily updated [19,90,91,99]. An eager update scheme ensures the root always reflects the most recent state of the memory, which requires updating the whole MT branch with the root for each encryption counter update. Such an update scheme incurs multiple memory accesses for each update, and significantly degrades the performance. Despite the performance degradation, it allows recovery after a crash in case of recovery expectation (NVM main memory) if all the updates were persisted. In case of the MT were eagerly updated but the updates were not strictly persisted, the BMT can be rebuilt if only the encryption counters are persisted, but the nodes inter-dependencies of the ToC makes it impossible to recover from the encryption counters [15, 99].

The lazy-update scheme updates only the encryption counter and relies on the natural eviction to propagate the updates upwardly [15, 19, 99]. Whenever a dirty node is evicted, its parent is fetched and updated. Such an update scheme reduces the memory accesses and the performance overheads

significantly, but will have a stale root value, and relies on the cached security metadata to represent the most recent state of the memory. Thereby, systems implementing a lazy-update scheme are performance friendly, but more susceptible to crash consistency problems [19, 55, 90].

Fabric-Attached Memory (FAM)

FAM architecture differs from traditional processor centric architecture by disaggregating the memory from the processing unit, and implements a shared large memory pool. The memory pool can be accessed directly by any processing element without the need to go through a home processor as in NUMA systems [47]. The main enabler of the FAM architectures are FAM protocols such as Gen-Z [5], CCIX [2], and CXL [3]. FAM protocols which are being currently developed by the joint efforts of leading system providers such as Google, HP, IBM, Dell EMC, Micron. Such protocols, require the processing elements to implement the memory semantic protocol at the memory controller to access the shared memory pool [5].

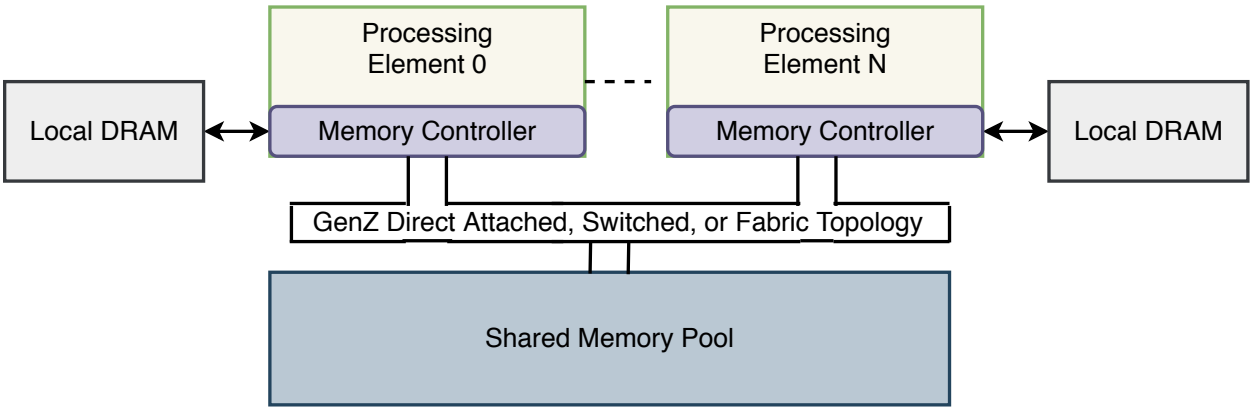

Figure 4.4: Fabric-Attached Memory Architecture.

FAM architecture, shown in Figure 4.4, has several advantages over traditional NUMA systems. For instance, for a processing element to access data in a different node's memory, the request has

to go through the home processor of that specific memory, which is typically done using expensive message passing protocols. In FAM architecture, sharing data does not require moving the data from one node to another, or sending a request to home processor, data sharing is a matter of assigning the memory region to the requester.

Motivation

In FAM architecture, each node has an access to a local DRAM and an access to a global shared memory pool. The local DRAM is used to cache the global memory data to improve the system performance. Therefore, implementing secure memory requires special handling, due to having two separate memories. Implementing integrity verification using a single Merkle-Tree as used in traditional systems can incur high overheads. Moreover, a single Merkle-Tree covering the global shared memory and the private local memories of all the nodes in the system can significantly reduce the global NVM lifetime. Additionally, as the Merkle-Tree nodes are required to be updated atomically with the data, this atomicity requirement can lead to even higher overheads, as writing a cacheline at any node's local memory will require locking the Merkle-Tree branch covering the modified cacheline until the atomic Merkle-Tree branch update is performed. Moreover, using a single Merkle-Tree will require persisting the whole Merkle-Tree branch and the updated node atomically, which will cause the local DRAM to operate in a write-through manner.

To reduce the overheads of secure memory implementations, a Split-Tree scheme can be used. In the Split-Tree scheme, a Merkle-Tree is used for the global memory, and another Merkle-Tree is used to protect the local DRAM. However, using two different trees can introduce caching problems as the security metadata for the trees will be contesting over the cache resources. Thus, caching the security metadata for both memories can cause a contention over the cache resources and lead to unnecessary overheads. For instance, the global memory is only accessed when the re-

quired data are not present in the local memory, but the global memory Merkle-Tree is expected to be huge. On the other hand, the local memory which is frequently accessed have smaller Merkle-Tree. However, the access latencies for the global memory is much higher than the access latencies for the local memory, and for the local memory security metadata to contest the cache resources with the global memory can degrade the system performance. Moreover, security metadata caching can have drastically different behavior depending on the used Merkle-Tree update scheme. As using an eager-update scheme will cause Merkle-Tree nodes in higher levels to be always cashed, as they will be used more frequently. Once the global memory is put into perspective, global memory security metadata cachelines are highly likely to be evicted due to the less frequent accesses.

On the other hand, using a lazy-update scheme tend to cache the lowest levels of the Merkle-Tree and the encryption counters, as it does not require to update the whole tree path for each access. However, using a lazy-update scheme can lead to crash consistency problems in systems with recovery expectations. As FAM architectures are expected to use NVMs as the shared memory pool, lazy-update scheme is not suitable for such systems. To address these challenges, we propose Split-Tree, a scheme that uses separate integrity trees to secure FAM architecture, and provides a security metadata cache partitioning scheme that studies the trade-offs and performance overheads of caching security metadata in FAM architectures.

Design

Overview

In secure FAM architecture each processing element is expected to have a memory region of the shared global memory as well as a small local memory. Therefore, each processing element is responsible for implementing data confidentiality and integrity support for its associated memories.

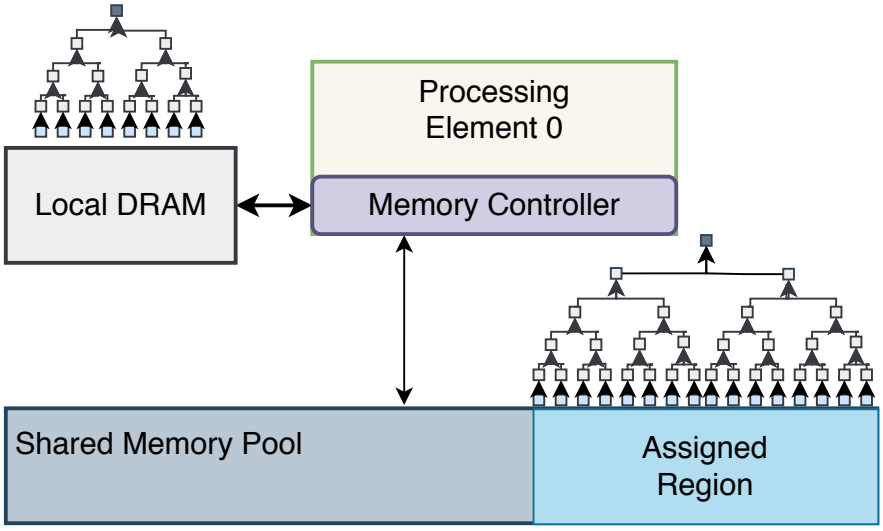

Figure 4.5: Split Merkle-Trees Design.

Since the local memory is used to cache the global region data, a naive approach would be to apply the encryption and integrity verification for the global memory, and use the same encryption counter and integrity tree to verify the integrity of the local memory data. However, the size of the security metadata responsible for protecting such large memory region is expected to be huge. For instance, to protect a memory region of 1TB using split-counter mode encryption and BMT, it would require 16GB for encryption counters and 11 levels BMT of about 19GB. Having an integrity tree with this huge size will make integrity verification a very costly process, even when caching is used. Therefore, we propose a design that has two separate Merkle-Trees, a large one used to protect the integrity of the owned global memory region, and a smaller one protecting the local memory. We refer to the shared memory region Merkle-Tree as global MT, and to the local memory Merkle-Tree as the local MT.

Figure 4.5 shows the implementation of the two Merkle-Trees. The Split-Tree scheme has several advantages over the single Merkle-Tree scheme. For instance, the shared memory region should be encrypted using different encryption counters and different processor key to allow data sharing,

and preserve each processing element private data. As if the shared region needs to be re-assigned to a different processing element, the new global region owner needs to posses the used encryption key, have access to the most recent encryption counters, and the most recent integrity tree root. Using a single Merkle-Tree means all the processing elements in the system should have the same encryption key. In case of a malicious processing element, or if an attacker obtains access to one processing element, the attacker can access any data in the system that belongs to different nodes. Therefore, using the same encryption key will compromise the security of the other processing elements in the system. Moreover, as the encryption keys are different, the Merkle-Tree protecting the global region is different from the one protecting the local memory to enable data sharing while preserving the security of each processing element. Data sharing can be enabled by passing the region encryption key to the requester along with the Merkle-Tree root, which can be done securely as described in earlier work [72, 73].

Additionally, resolving any security metadata cache miss will require accessing the global memory. However, the global memory access latencies are expected to be $\tilde{3}$00ns for reads and $\tilde{1}$000ns for writes [47]. On the other hand, having a Local MT can obtain the security metadata from the local DRAM.

Moreover, having two different Merkle-Trees can reduce the writes to the NVM global memory by updating the local MT when a data cacheline is evicted from the processor caches to the DRAM. Thus, increases the NVM lifetime which has a limited write endurance. However, having two different Merkle-Trees and two different set of encryption counters requires decrypting/re-encrypting the data when transferred between the different memories.

Data Transfer Between Memories

Each node in FAM architecture is expected to have a local memory used to cache the data from the assigned global memory region. For performance reasons, the local memory is expected to be a DRAM, but the global memory is expected to be a NVM for higher capacity, persistency and lower power requirements. As the DRAM is used to cache the global memory region, data transfer between the local memory and the global memory is expected to be managed by an extension of the memory controller as in Intel Xeon scalable processor memory controller for Intel Optane DC-Memory mode [8]. Having a single Merkle-Tree to protect the memory integrity will require the encrypted data to be migrated from the global memory to the local memory, and then perform the decryption when the data is fetched from the local memory to the processor cache hierarchy. While this scheme can simplify the implementation of security measures, it requires fetching the whole Merkle-Tree branch to verify the integrity of the required data. Moreover, updating the Merkle-Tree can be expensive in terms of performance overhead, extra writes to the global memory, and NVM lifetime.

On the other hand, using split Merkle-Trees requires decrypting the data as it gets migrated from the global memory, and re-encrypting it as it gets inserted into the DRAM, which can be done at the memory controller responsible for page migration.

Figure 4.6 shows how the re-encryption process is performed when a page is migrated from the global memory region to the local memory. In Step 1, the memory controller initiates a page migration from the global memory to the local memory, which requires allocating a physical frame in the local memory for the requested page. While the page is being fetched from the global memory, the memory controller generates the global memory page OTPs and the allocated local memory OTPs by notifying the AES encryption engine as shown in steps 2 and 3. When the data arrives from the global memory, the encrypted data is XORed with the global memory OTP to

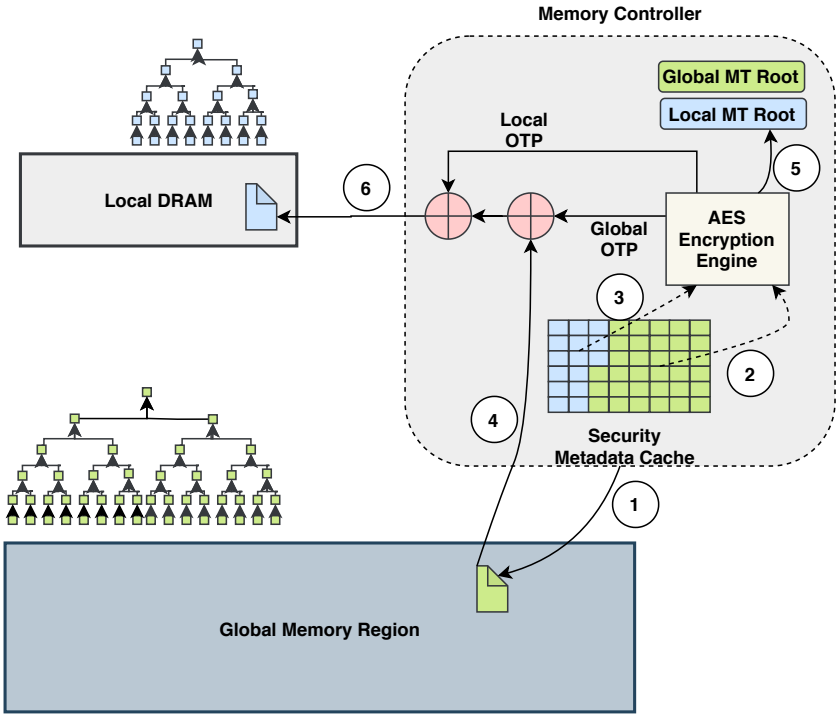

Figure 4.6: Split Merkle-Trees re-Encryption.

complete the decryption. After that, the decrypted data is XORed with the local memory OTP to complete the local memory encryption as shown in Step 4. After the re-encryption is done, the local MT root is updated in Step 5. Finally, the data is sent to the local memory in Step 6. Note that generating the OTPs in a timely manner requires the security metadata to be cached. However, caching security metadata can be problematic as the global MT is huge, yet the accesses to the global memory are less frequent than the local memory. Therefore, the local MT nodes will be replacing the global MT nodes in the cache due to the LRU replacement policy.

Caching Security Metadata

Due to the frequent accesses to the local memory, the security metadata of the global MT will be evicted from the security metadata cache, resulting in multiple accesses to verify the integrity for the global memory to resolve a miss. However, as the local memory is a DRAM, it does not require a recovery mechanism. Therefore, using a lazy update scheme to update the local memory Merkle-Tree can reduce the number of accessed metadata for each update. On the other hand, as the global memory region is a NVM, the system should be able to verify the integrity of the NVM data after crashes. Therefore, we use an eager update scheme to update the global memory. Note that, data is written to the global memory region when it gets evicted from the local DRAM.

While using a lazy-update scheme to update the local Merkle-Tree can prevent the local MT nodes from evicting the global MT nodes, but it may lead to the global MT nodes evicting the local MT nodes due to the eager update scheme used for the global MT. Therefore, we partition the security metadata cache to prevent premature eviction of the security metadata cachelines. While partitioning the cache can prevent evicting the security metadata of the global MT, it would still have to access the global memory to fetch the required metadata. To reduce the global memory region accesses, we use a small unprotected region in the local memory to cache the security metadata of the global memory region. Thus, whenever a global memory security metadata cacheline is evicted from the cache, the block is written back to the unprotected memory region as well as the global memory region. Caching the global memory region security metadata in the DRAM reduces the access time of the required metadata to a DRAM access latency instead of the global FAM region. Figure 4.7 shows the global MT is updated when a dirty page is written back from the DRAM to the global memory region. In Step 1, the memory controller selects a page in the DRAM to be replaced. After that, the memory controller reads the security metadata of the page from the unprotected region in the DRAM and uses the security metadata to generate the OTPs as in steps

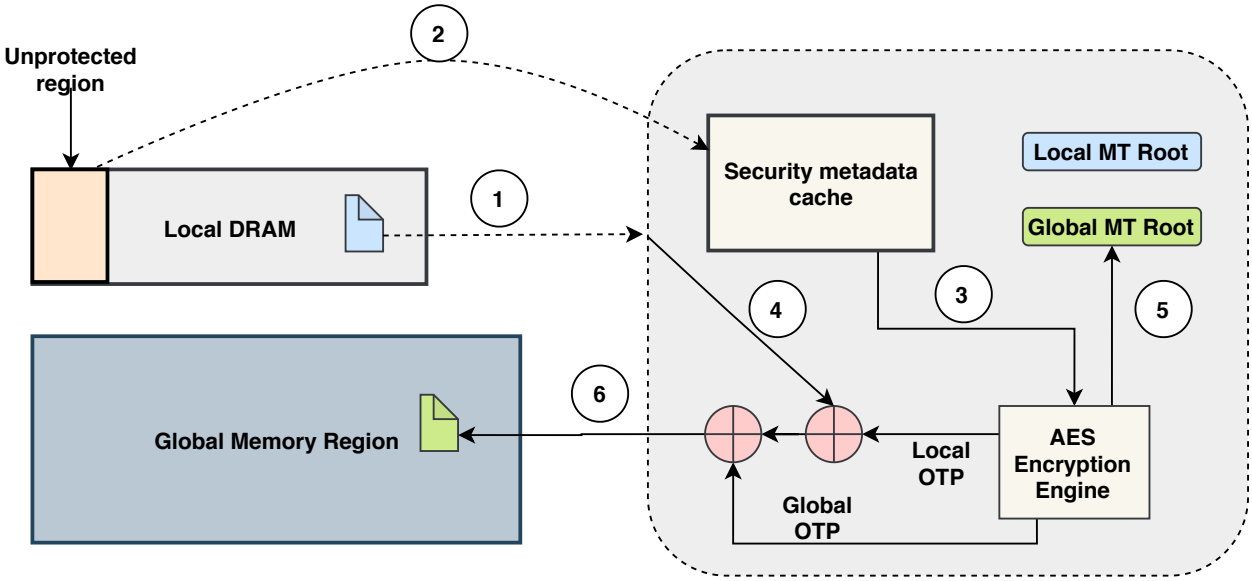

Figure 4.7: Updating Global Memory Region Merkle-Tree.

2 and 3. In Step 4, the data is decrypted using the local memory OTP and re-encrypted using the global memory OTP. Finally, the global MT root is updated and the data is written back to the global memory region in steps 5 and 6. Note that the global memory region security metadata are only updated when the new global OTP is generated, which requires updating the whole global MT branch and the root. Finally, writing the data back to the global memory region (NVM) should be done in an atomic manner, in which the data is written atomically along with its associated security metadata. Write atomicity can be achieved by utilizing the Write Pending Queue (WPQ), which is a persistent buffer in the memory controller. The WPQ is supported by the Asynchronous DRAM Self-Refresh (ADR) which provides enough power to flush the WPQ contents in case of a crash.

Design Discussion

In this Section, we discuss some design options and the overheads of our scheme.

The overhead of the re-encryption process is minimal as counter-mode encryption scheme is used, the OTPs generation time can be overlapped with memory read time, and XORing the encrypted data with the OTP completes the decryption. Therefore, the overhead of the re-encryption process is limited to few cycles required to perform the XOR operations.

Both BMT and ToC can be used to provide the required integrity verification. However, in our scheme we use a BMT to protect the integrity of the local memory and a ToC for the global memory. The BMT does not allow a parallel update, but the leaves of the BMT has a higher coverage than ToC leaves, as each leaf node in the BMT covers one page (4KB) of data. On the other hand, the global memory region requires recoverability measures and thus an eager update scheme is required. We use a ToC due to the performance advantage by allowing parallel updates.

As the security metadata of the global memory region is cached in the local DRAM, the DRAM region used for caching is unprotected for two reasons. First, protecting the caching region will require updating the local memory MT and encryption counters each time a cacheline is written, which can introduce unnecessary overheads. Second, the encryption counters are already protected using the Merkle-Tree, which has its root stored securely in the processor.

Security Discussion

The security of the memories is protected using the encryption counters and the Merkle-Trees. As the re-encryption process is performed inside the secure region, our scheme does not affect the security of the system. However, to ensure the security of the counter-mode encryption scheme, encryption counters re-use is prohibited. Therefore, in a post-crash situation the encryption counters for the local memory (DRAM) are reset, which can open a room for encryption counters reuse. Thus, the encryption key used to encrypt the local memory data should be changed after each crash. On the other hand, changing the encryption key for the global memory region (NVM)

is not required as the encryption counters are persisted.

Note that passive attacks aiming to read the data are not possible, as the data is encrypted in the memories and the bus. Active attacks trying to tamper with the data or replaying old packets are detected using the Merkle-Trees, and thus will fail the integrity verification.

Finally, assigning global memory region to a processing element requires transferring the encryption key from the processing element managing the global memory to the requester. The key exchange process should be done in a secure key exchange mechanism as used in previous work [21] to exchange keys between the processor and the memory.

Methodology

To evaluate our scheme, we used the Structural Simulation Toolkit (SST) [70]. We implemented the BMT and the ToC, and the security metadata caches. We added latencies to model the overhead of encryption. We modified the memory controller to handle the encryption and integrity verification. The configuration of the modeled system are listed in Table 5.2.

To stress our proposed scheme, we used memory intensive applications from SPEC2006 [37] benchmarks, and some HPC workloads such as Lulesh2.0 [42], miniFE.x [38], pennant [33], and SimpleMOC [35]. We run each application for 500M instructions using a single thread for SPEC2006 applications and four threads for other workloads. We run the experiments using a single processing element as using multiple processing elements sharing the memory can result in security metadata coherence problem, which is beyond the scope of this work. We implemented a baseline scheme that uses a single Merkle-Tree protecting the global memory region and does not protect the local memory. We implemented a scheme that uses a single Merkle-Tree to protect both memories and does not partition the security metadata cache. Then, we compared the single-MT

Table 4.1: Configurations of the Simulated System.

Processing Element (PE)	
Processor	4 Cores, X86-64, Out-of-Order, 2.00GHz
L1 Cache	Private, 4 Cycles, 32KB,8-Way
L2 Cache	Private, 6 Cycles, 256KB, 8-Way
L3 Cache	Shared, 12 Cycles, 1MB/core, 16-Way
Cacheline Size	64Byte
Fabric latency	40 ns
Memory	
Local Memory	256MB DRAM
Global Memory Region Capacity	16GB
NVM Latencies	Read 300ns, Write 1000ns
Encryption Parameters	
Security Metadata Cache	256KB, 8-Way, 64B Block

scheme and our proposed split Merkle-Tree scheme with the baseline. Finally, we implemented the cache partitioning and used a DRAM cache region.

Evaluation

To evaluate our scheme, we implemented the Split-Tree scheme, a Single-MT scheme, DRAM caching for global MT nodes, and security metadata cache partitioning. We compared the results with a baseline of a system that only protects the global memory with one tree.

Split-Tree Impact on Performance

Figure 4.8 shows the performance overheads of Single-MT scheme, and Split-Tree scheme compared to the baseline. The Single-MT scheme has an average performance overhead of 14%, which spikes to reach 51% for *cactus*, 34% for *mcf*, and 25% for *lbm*. This spike can be explained by the

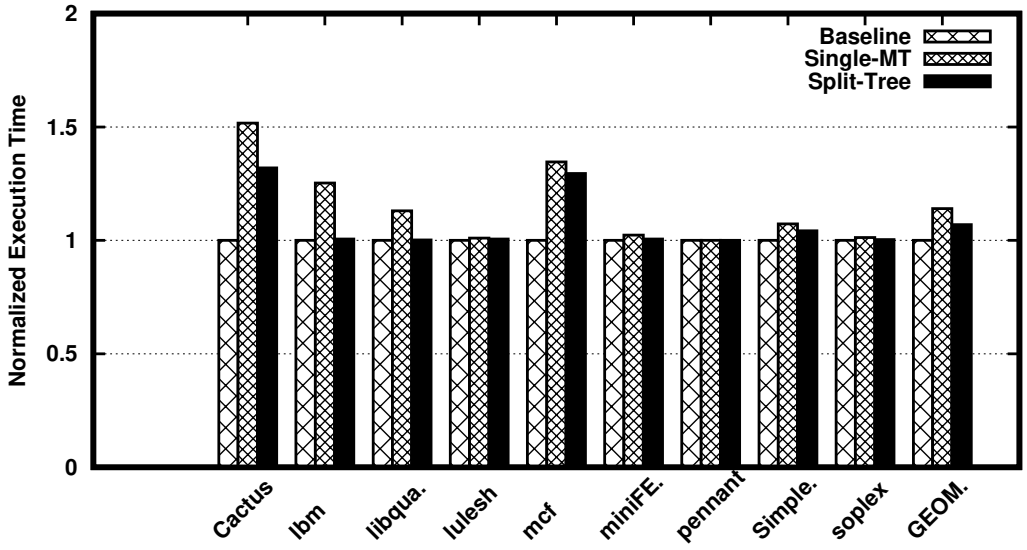

Figure 4.8: Split-Tree Impact on Performance.

increase in global memory requests observed by these applications in the Single-MT scheme as shown in Section4 and Section 4. On the other hand, the Split-Tree scheme has an average performance overhead of 6.9% which spikes to reach 31% for *cactus*, and 29.5% for *mcf*. The overheads are caused by the Split-Tree local memory accesses as discussed in Section 4 and Section 4. Note that the Split-Tree scheme is reducing the performance overhead by making the requests going to the local memory which has faster accesses than the global memory.

Split-Tree Impact on Memory Reads

As shown in figure 4.9, the Single-MT scheme has an average of 34.5% reads to the global memory. These reads are caused by the security metadata misses which are required to verify the integrity of the required data cachelines. On the other hand, the Split-Tree scheme has no extra reads to the global memory, in a matter of fact, Split-Tree scheme reduces the global memory reads by an average of 1%, which is caused by verifying the integrity of the required data cacheline using the

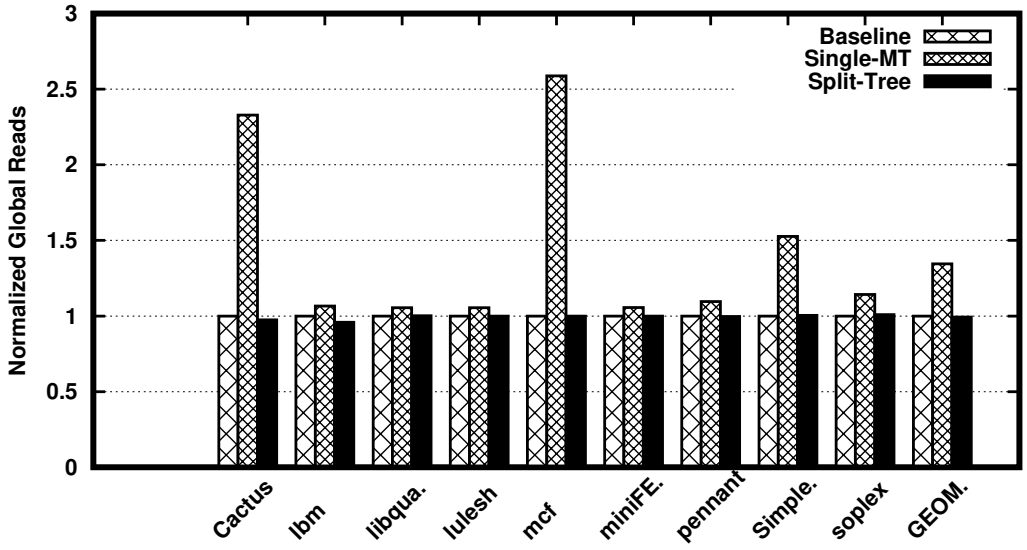

Figure 4.9: Split-Tree Impact on Global Memory Reads.

local memory MT. However, the Split-MT scheme reduces the global memory reads by increasing the local ones.

As shown in figure 4.9, the Single-MT global memory reads spikes to reach 259% for *mcf*, and 232% for *cactus* which explains the high performance overhead for these applications. On the other hand, Split-MT scheme has 5% less global memory reads for *lbm* which explains the huge performance improvement for this application in the Split-MT scheme.

Figure 4.10, shows the normalized read operations by the schemes. We observe that Single-MT scheme has no effect on the local memory reads, which is expected as the Single-MT scheme protects the local memory using the same MT protecting the global memory, and as a result the security metadata misses are fetched from the global memory as discussed earlier.

The Split-MT scheme has 193% local memory reads on average, which spikes to reach 644% for *cactus*, 446% for *mcf*, and 383% for *SimpleMOC*. The high number of local memory reads for these applications is explained by the low security metadata cache hit rate for these applications

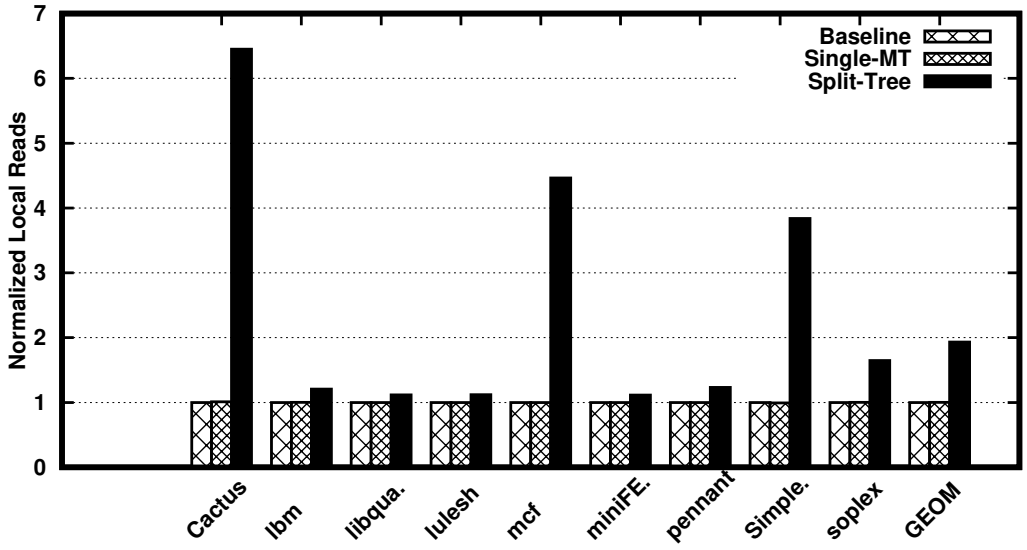

Figure 4.10: Split-Tree Impact on Local Memory Reads.

as discussed in Section 4. However, we notice that *SimpleMOC* does not have a high performance overhead as other applications having similar memory accesses, which is explained by the low number of memory accesses for this application.

Split-Tree Impact on Memory Writes

Figure 4.11 shows the global memory writes incurred by the schemes. The Single-MT scheme has an average of 241% writes, which spikes to reach 480% for *mcf*, and 365% for *SimpleMOC*. We observe that applications having the lowest security metadata cache hit rate are experiencing the highest extra memory accesses, and due to protecting the local memory using the same MT protecting the global memory, the accesses to fetch the required security metadata are directed to the global memory. On the other hand, the Split-Tree scheme has no extra writes to the global memory but reduces the writes to the global memory by 1.1% on average. We observe that *lbm* has 10% less writes to the global memory, which is caused by having a higher security metadata

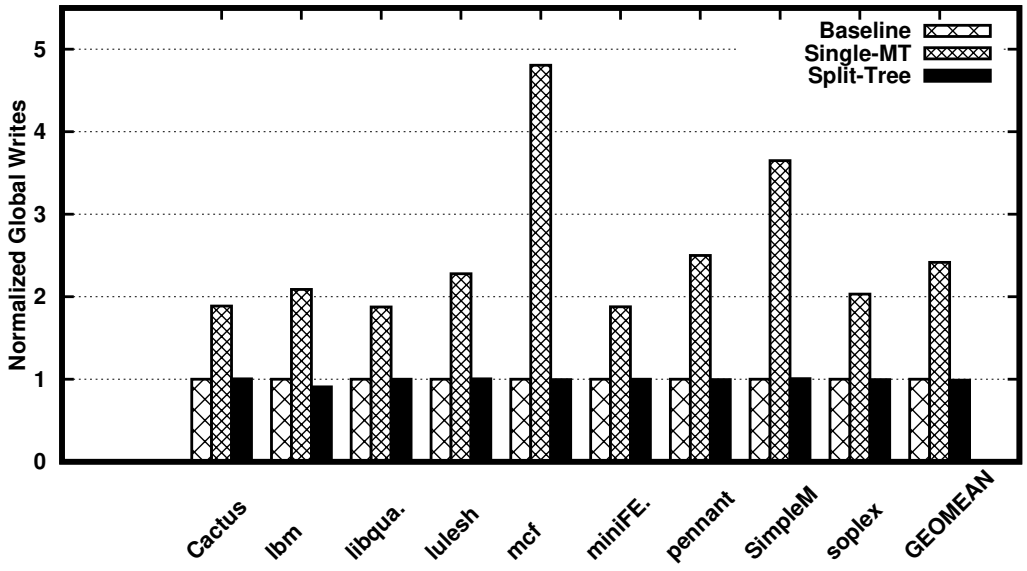

Figure 4.11: Split-Tree Impact on Global Memory Writes.

cache hit rate in the Split-MT scheme.

Figure 4.12 shows the local memory writes for the schemes. We observe that Single-MT scheme has no effect on the local memory writes as the security metadata writes are directed to the global memory. On the other hand, the Split-MT scheme has an average of 29% extra writes to the local memory which spikes to 224% for *mcf*, and 220% for *SimpleMOC* due to the low security metadata cache hit rate. We noticed that Split-MT scheme has two advantages over the Single-MT scheme in terms of writes. First, Split-MT changes the writes overhead to the local memory (DRAM) instead of the global memory (NVM), which translates into better performance and increases the lifetime of the NVM by 2.4x. Second, as the local memory has a smaller capacity and thus a smaller MT, the local MT updates are causing only 29% instead of the 124%.

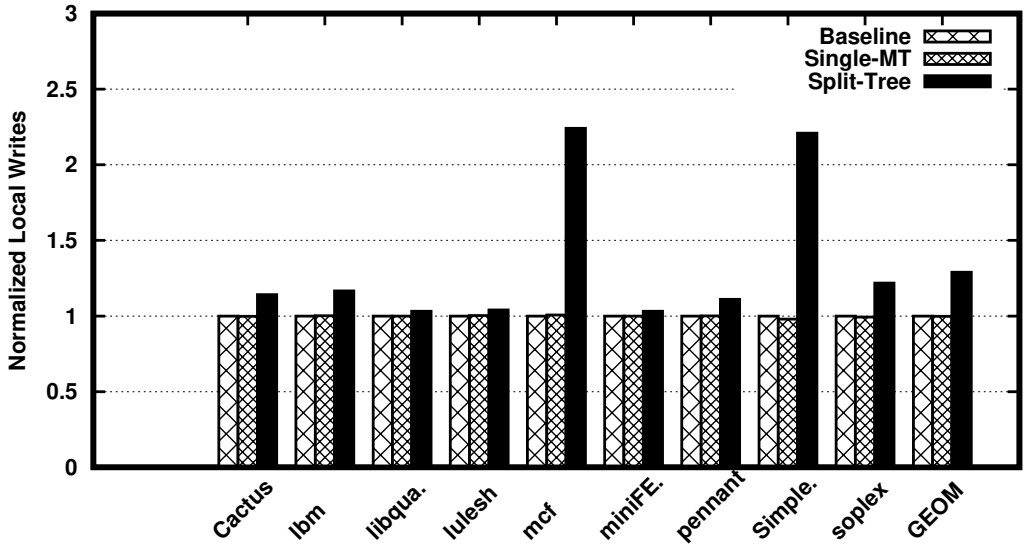

Figure 4.12: Split-Tree Impact on Local Memory Writes.

Cache Partitioning Impact on Split-Tree

In order to analyze the effect of Split-Tree on the security metadata cache, we started by collecting the security metadata cache hit rate for the Single-MT scheme, the local MT in the Split-Tree scheme, and the global MT in the Split scheme, and finally the overall Split-Tree scheme.

Figure 4.13 shows the security metadata cache hit rate for the schemes. Single-MT scheme has an average security metadata cache hit rate of 80%, with *cactus* having a 48% hit rate, *mcf* and *SimpleMOC* having a 65.2% hit rate. On the other hand, the global MT in the Split-Tree scheme is showing a better security metadata cache hit rate for all the applications with an average of 85% hit rate. The hit rate of the MT is improved because the global MT is actually smaller than the Single-MT and it is only updated whenever data is written back to the NVM. However, the local MT is showing a very low average hit rate of 58.8% where *cactus* is having a 16.3%, *mcf* is having 29.5%, and *SimpleMOC* is having a 34.1% hit rates. Using a BMT for the local MT which is lazily updated makes the usage for the local MT nodes less frequent comparing to the eagerly updated

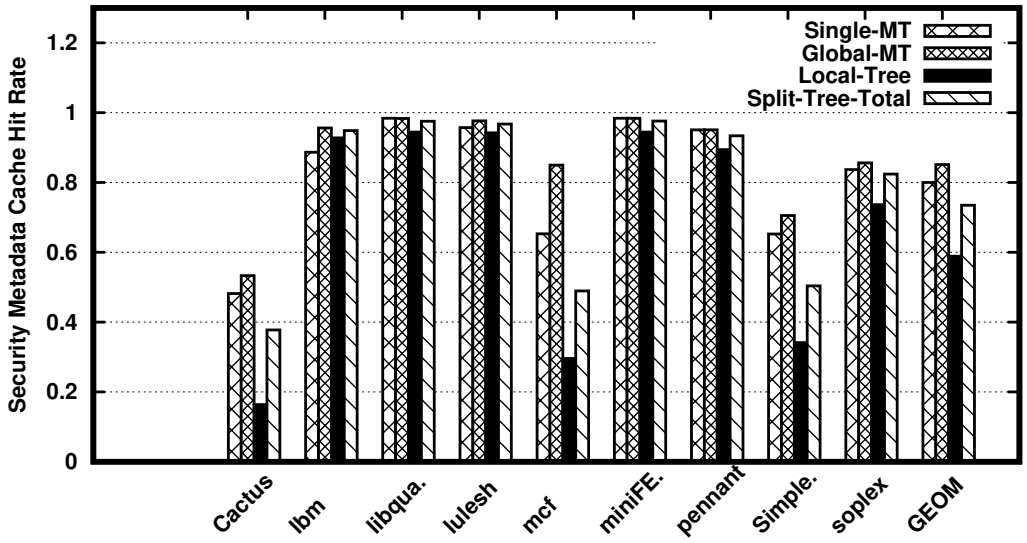

Figure 4.13: Security Metadata Cache Hit Rate.

global MT which results in evicting most of the local MT nodes. While caching the global MT nodes is more desirable but it is severely degrading the hit rate for the local MT. To analyze this low hit rate of the local MT, we collected the distribution of security metadata cache and analyzed the accesses to the MT nodes. We observed that without partitioning the security metadata cache, the global MT nodes are occupying 97.8% of the security metadata caching.

Figure 4.14 shows the access distribution for the MT levels. We observe that Single-MT and the global MT are showing a similar behavior where 25.5% and 29.5% of the accesses are going to the first level, then the number of accesses are saturating around 11%. The saturation level 11% represents the generated writes used to eagerly update the trees. The lower levels are showing higher accesses due to stopping the verification of read data at the first cache hit. On the other hand, the local MT is lazily updated and thus the higher MT nodes are only required when a dirty child node is evicted, or for read verification of the child node is missing. Due to this less frequent of the upper nodes they get evicted by the contesting eagerly updated global MT nodes. However, this is not the case with the Split-Tree scheme. As shown in Figure 4.14, the local MT has 4 levels

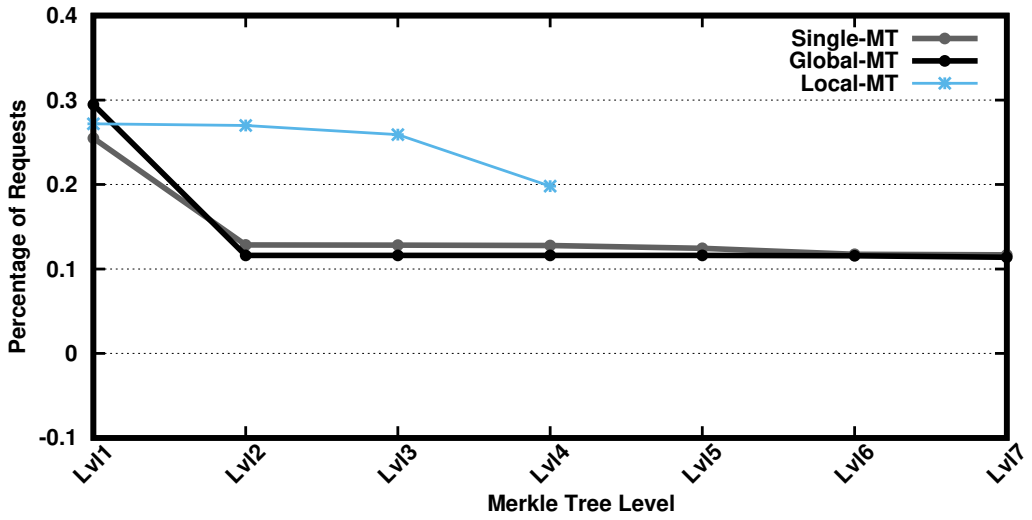

Figure 4.14: Access Distribution to Merkle-Tree Levels.

which are receiving 27%, 26.9%, 25.9%, and 19.8%. Despite the use of the lazy update scheme, and the high node coverage of the BMT, the local MT lower levels nodes are getting evicted which requires requesting higher levels for read verification, which explains the access behavior of the local MT levels despite the use of the lazy update.

DRAM Metadata Caching and Cache Partitioning Impact on Split-Tree

To improve the system performance, we enabled DRAM caching of global MT nodes by allocating a 256KB region and using it as a cache. The DRAM caching improved the system performance slightly. Note that, a DRAM cached metadata miss will cause a higher latency to fetch the required global MT node, as the memory controller has to check the DRAM caching region first and then send the request to the global memory if the node is not cached in the DRAM cache region. To improve the performance further, we overlapped the requests by sending the request to the local memory as well as the global memory, and if the request was served from the DRAM, the global request is ignored. Finally, we applied cache partitioning techniques to prevent the local MT and

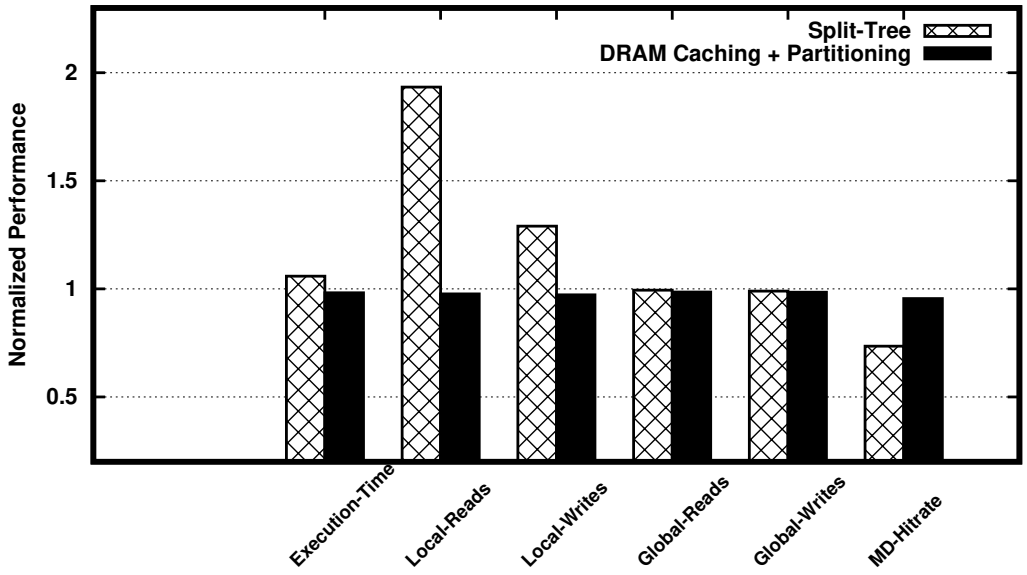

Figure 4.15: DRAM Caching and Partitioning Impact on Performance.

the global MT from contesting over the cache resources. As the local MT is using a lazy update scheme and due to the small local memory size, we limited the local MT nodes partition to 30% of the security metadata cache size. Furthermore, we cached the lowest two levels of the local MT only and partitioned the global MT cache sets to prevent lower global MT levels from evicting higher MT levels, and assigned a smaller number of sets for higher levels. The MT node coverage increase as the level of the MT node increases, therefore we assigned one set for the highest two levels, and increased the number of sets for the lower levels dynamically.

Figure 4.15 shows all the performance aspects of the DRAM caching combined with cache partitioning compared to the Split-Tree. The cache partitioning and DRAM caching of security metadata improved the performance by 7%, local memory reads by 50%, local memory writes by 30%. We observe that, despite the use of DRAM caching which is expected to increase the number of local memory accesses, the number of accesses to the local memory decreased. This decrement is justified by the huge improvement in the security metadata cache hit rate, as the cache partitioning

improved the hit rate from 73.5% to 95.5%.

Related Work

Secure memory implementation has been studied from different perspectives by variety of studies. Osiris [90] discussed the crash consistency problem of secure NVMs, and highlighted that a power failure or a crash can result in having stale encryption counters, which can lead to integrity verification failure, and thus losing the whole memory content. Osiris proposed a scheme to recover the encryption counters after a crash, which relies on a stop-loss mechanism coupled with using ECC bits as a sanity check for the recovered counter correctness. Anubis [99] addresses the recovery time problem in secure NVMs. Anubis emphasizes that recovering the encryption counters is not always sufficient to recover the integrity tree. Moreover, rebuilding the integrity tree can take hours for practical size NVMs. Therefore, Anubis proposed a scheme that tracks the updated security metadata cachelines in the cache. During the recovery phase, Anubis relies on the tracking mechanism to pin-point the lost data and recover it. Phoenix [15] highlights the overheads caused by Anubis scheme when a ToC is used. Phoenix [15] aims to reduce the number of writes incurred to recover the ToC by utilizing an encryption counters recovery scheme to recover the encryption counters, and tracks the updates of unrecoverable intermediate ToC nodes. VAULT [81] discussed the overheads caused by the integrity tree and proposed a scheme to reduce these overheads. VAULT proposed having a variable arity tree, in which lower integrity tree levels can pack more child nodes and the arity decreases as we go higher, until it saturates at an arity of eight. VAULT reduces the depth of the integrity tree and thus reduces the number of accesses required to verify the integrity of an encryption counter or update it. Synergy [77] discussed the overheads of Message Authentication Codes (MACs) associated with the data in secure memory architectures. As counter mode encryption is used to protect the data confidentiality, and Merkle-

Tree is used to verify the encryption counters integrity, the data integrity is protected by calculating a MAC value over the data and the encryption counter. Therefore, protecting the encryption counters is sufficient to ensure the integrity of the data, due to the attacker inability to generate the same MAC values [71]. Synergy [77] aims to reduce the number of memory access required for integrity verification by replacing the ECC bits with MAC value, and storing the ECC bits in the memory instead. Synergy relies on the fact that MAC values can be used for error detection as well, and will always be required for integrity verification. On the other hand, ECC bits are used for error checking and rarely used for error correction. Rogers et al. [72] proposed a scheme for data protection in Non-Uniformed Memory Access (NUMA) systems. The proposed scheme assumes each node is protecting its memory, which leaves the interconnects and message communication between nodes unprotected. To protect the interconnects and the communicated messages, a point-point encryption is used. The encrypted messages are associated with MAC values to ensure their integrity. Morphable counters [76] discussed the overheads of secure memory implementation, and suggested that increasing the encryption counters cacheability can increase the cache hit rate and improve the performance. Morphable counters proposes a scheme that allows packing a maximum of 128 encryption counters per cacheline, but reduces the counters size and uses some bits for the management. However, using small counters can cause frequent minor counters overflow which can lead to the whole page being re-encrypted. Therefore, Morphable counters discusses the trade offs between counters cacheability and the overflow.

Conclusion

Protecting the confidentiality and integrity of the data in FAM architecture is challenging and requires special handling due to having two different memories. Implementing secure memory architecture schemes directly can introduce higher overheads. Split-Tree is a scheme that uses a

dedicated integrity tree to protect the local memory, and another integrity tree to protect the global memory. Using two different integrity trees can reduce the performance overhead of traditional secure memory implementation schemes. However, having two different trees will cause a high contest over the security metadata cache and can lead to unnecessary performance overheads and extra memory accesses. To reduce the effect of the contest over the cache resources, we partition the security metadata cache to have static partition for the local MT and another partition for the global MT. Furthermore, we prevent caching higher levels of the local MT due to the use of lazy update for the local MT, and we partition the global MT cache sets between different MT levels dynamically.

Using Split-Tree can reduce the performance overhead by 7%, global memory reads by 34%, global writes by 140%. However, this improvement is achieved by increasing the local memory reads by 93%, and local memory writes by 29%. Finally, implementing cache partitioning techniques and allowing DRAM caching of the global MT nodes improved the performance by additional 7%, which is stemming from the high improvement of the security metadata cache hit rate.

CHAPTER 5: MINERVA: RETHINKING SECURE ARCHITECTURES FOR THE ERA OF FABRIC-ATTACHED MEMORIES

Background and Motivation

In this Section, we discuss the most related concepts followed by the motivation of our work.

Background

Threat Model

In this work, we assume an attacker capable of passive and active attacks. Similar to state-of-the-art work in secure memory architecture [19, 76, 77, 81, 90, 91, 99], the attacker is capable of scanning the memory to read its content, snoop the memory bus, drop packets, tamper with the packets or memory content, and replay old packets. Finally, memory access pattern leakage [85, 86], timing side-channel leakage [86], power analysis [59], and malicious PE(s) are beyond the scope of the work. However, our proposed secure architecture scheme is orthogonal to defenses targeting these types of attacks, thus it can be integrated along with defenses against these attacks to achieve a higher degree of protection.

Fabric Attached Memories

Current High-Performance Computing (HPC) systems, shown in Figure 5.1-A, are implemented such that each computeg node has its own memory module(s) attached to it directly. One of the main issues with these architecture is that memory modules are only utilized by the processor at-

tached to it, which can lead to large portions of the memory being underutilized since accessing data in other nodes is typically implemented through expensive network interfaces by leveraging message passing libraries [47]. This approach of coupling memory modules to processors limits the efficiency, flexibility and performance of current systems. Therefore, system vendors are moving toward memory-centric architectures such as HP's "The Machine" [43]. FAM architectures, shown in Figure 5.1-B, allow multiple processing elements to connect to a shared memory pool using a connecting fabric, and seamless integration of processing elements from different vendors. The main enabler of FAM architectures is the connecting fabric, which is governed by a connecting protocol such as Gen-Z [5], Compute Express Lanes (CXL) [3] or Cache Coherent Interconnect for Accelerators (CCIX) [2].

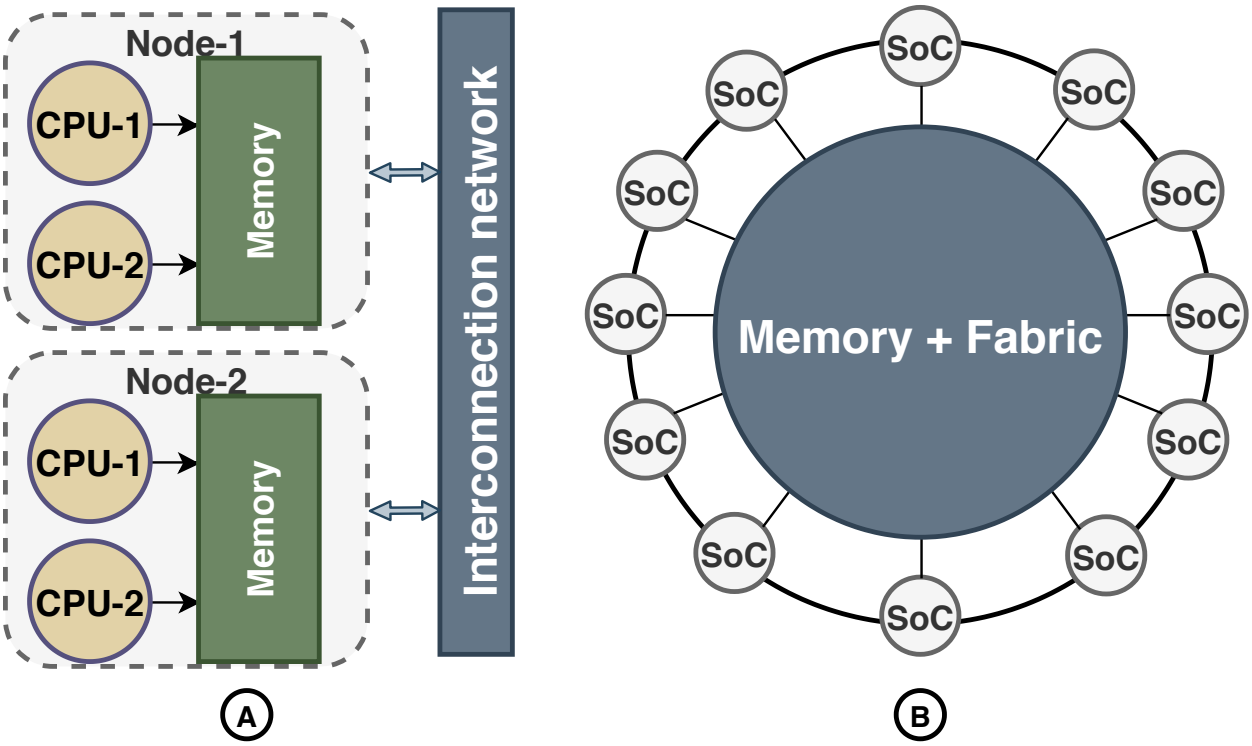

Figure 5.1: Conventional and FAM architectures.

FAM architectures are expected to have large shared memory pools and, due to the high power requirements for frequent refresh and cooling of DRAM, FAM architectures are expected to use

emerging NVMs as the foundation of the shared memory nodes [43].

Emerging Non-Volatile Memories (NVMs)

Emerging NVMs feature high density, near-zero idle power consumption, byte addressability, the ability to retain data during power loss, and access latencies comparable to DRAM. Due to these features, NVMs are considered a promising contender to partially or fully replace the DRAM as main memory, which can be used to host persistent applications data. [18, 19, 21, 76, 81, 90, 99]. On the other hand, NVMs suffer from power-consuming writes, limited bandwidth, and low write endurance, which can be improved by using deduplication, compression, and/or wear-leveling of the NVM writes [18, 31, 32, 52, 57, 84, 102]. Moreover, NVMs' ability to retain data during power loss facilitates data remanence attacks, therefore NVMs are typically coupled with security features [90, 99].

Secure Memory Architectures

Secure memory architecture limits the trust boundary to the processor chip, therefore the memory content is usually encrypted to ensure data confidentiality, and a Merkle tree (MT) is used to verify the data's integrity. Below we describe the state-of-the-art schemes in memory encryption (counter-mode encryption), and integrity verification (Merkle tree).

Counter-mode encryption is used in state-of-the-art processor systems as it provides strong defenses against a wide range of attacks. Namely, counter-mode encryption thwarts dictionary-based attacks, known plain-text attacks, and snooping attacks [18,19,88,90,99]. Moreover, counter-mode encryption reduces the decryption latency by overlapping the decryption with the memory read latency. To ensure the security of counter-mode encryption, reusing encryption counters (ECs) is

prohibited, as it facilitates known plain-text attacks [18, 19, 88, 90].

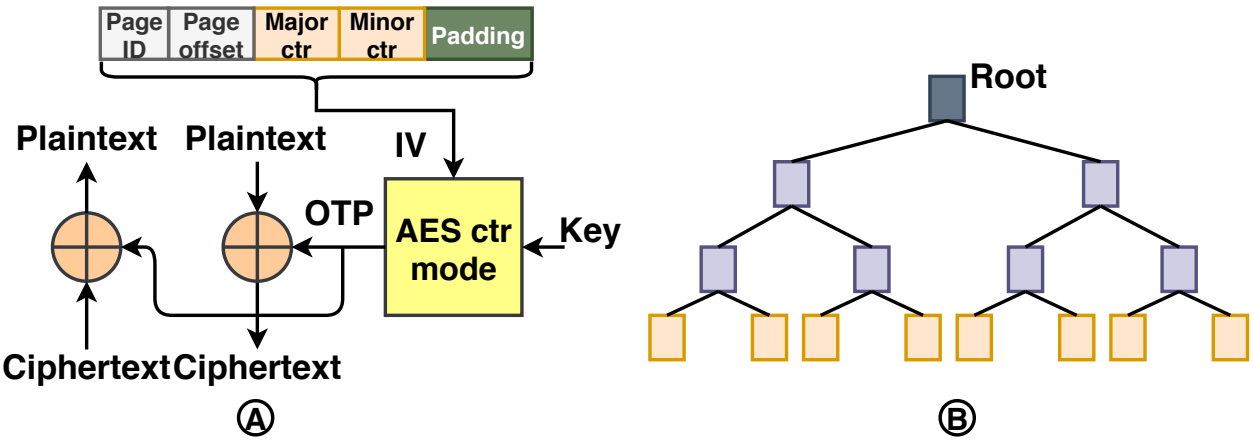

Figure 5.2: (a) Split counter-mode encryption, (b) Bonsai Merkle tree (BMT).

Figure 5.2:(a) shows the split-counter mode encryption and how it works. In split counter mode, an initialization vector (IV) composed of a per-page major counter, per-block (cacheline) minor counter, page offset, page ID, and padding is encrypted by an AES encryption engine using an encryption key to generate a One-Time-Pad (OTP). The OTP is then XORed with the plaintext/-ciphertext to complete the encryption/decryption [30, 55, 71, 77]. Each time a cacheline is written, the per-block minor counter is incremented and used to encrypt the cacheline. Whenever a minor counter overflows, the associated major counter is incremented and the whole page gets re-encrypted using the new counter. While counter mode encryption can protect data confidentiality, it cannot prevent replay attacks nor verify the data's integrity. Therefore, a Merkle tree is used to verify the data's integrity [15, 71, 91, 99].

Merkle trees (MT) are used in state-of-the-art schemes [19, 55, 81, 90, 99] for memory integrity verification purposes. Depending on the tree structure, Merkle trees can be non-parallelizable(e.g. Gneral Merkle tree) or parallelizable (e.g. SGX style counters tree). Additionally, the MT can be built on top of the encryption counters instead of the data, to reduce the MT size. Wherein the integrity of the encryption counters is protected by the MT, and the integrity of the data is

protected using keyed-Message Authentication codes (HMACs). This organization of the MT is typically referred to as the Bonsai-Merkle Tree (BMT) [71].

In general, the hashes of each level of the Merkle tree are calculated using the hashes of the level below, which requires a sequential update of the tree. Figure 5.2:(b) shows a 2-ary tree, as the figure shows, the bottom level which includes the encryption counters are hashed to generate the first level of the tree, then each 2 hashes from the first level are hashed together to generate the second level, the process continues recursively until a single node is generated, which is referred to as the root. To be able to use the root to verify the memory's integrity, it should be kept up to date in the secure region. To verify the integrity of a cacheline in a general MT integrity protected system, the hash of the cacheline is calculated. Then the calculated hash is used with the stored hashes to generate the root, if the calculated root matches the stored root, then the cacheline's integrity is verified. A faster way to verify the integrity of the cacheline is by stopping the verification process with the first parent hash cache hit, relying on the fact that a cached block's integrity is already verified.

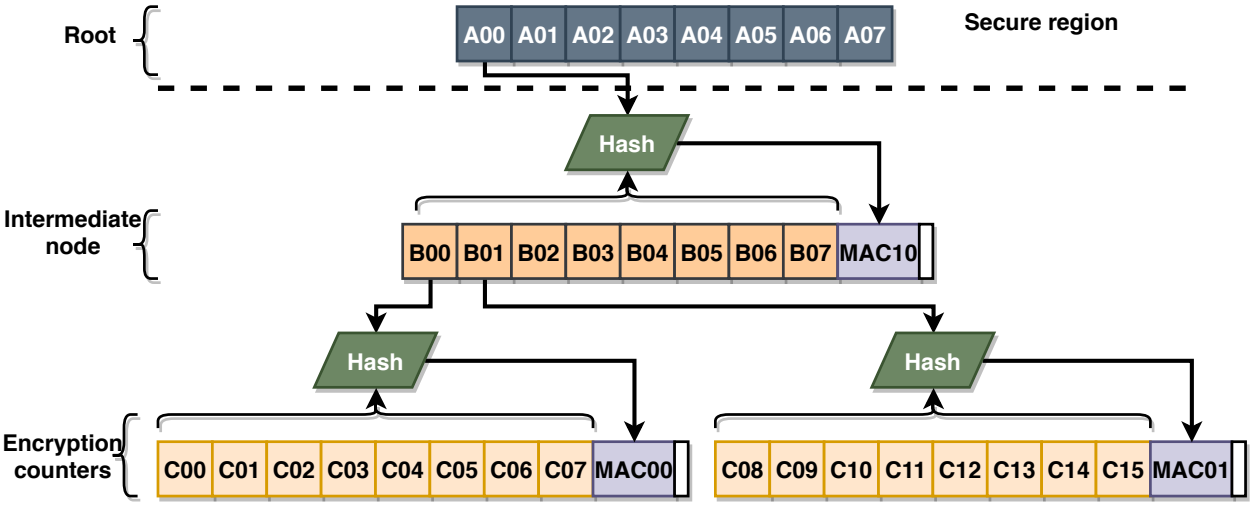

Figure 5.3: Tree of counters.

On the other hand, the parallelizable tree (known as Tree-of-Counters (ToC)) works differently;

ToC includes eight encryption counters and a Message Authentication Code (MAC) value in each leaf node. To generate the associated MAC value, the node counters, and a version from the parent level are hashed together. Figure 5.3 shows a subset of the ToC structure. Notice that, in ToC, whenever a data block is written back to the memory, its associated counter is incremented and the parent, also known as the version, of the counter is incremented as well [15, 30, 34, 99]. In ToC, the MAC value calculation does not depend on the below level as in BMT, thus the process of updating the tree can be done in parallel. Integrity verification of a memory block in ToC integrity protected system is done similarly as in BMT. Obviously, the tree size grows as the memory size grows, which can cause significant performance degradation to eagerly update the tree.

Integrity tree update schemes can either be eager or lazy [19, 99]. In an eager update scheme, the root always reflects the most recent state of the memory. Therefore, it should always be updated whenever a data block is written to the memory. On a data block write operation, the encryption counter of the data block is updated and the whole MT branch is updated until the root [19, 77, 81, 99]. This scheme allows the integrity verification to be done using a single hash value which is the root, and allows recoverability as the MT updates are persisted to the memory atomically with the data. However, it incurs multiple extra writes for each data write depending on the number of MT levels. In fact, as NVMs are expected to be in terabytes, eagerly updating the security metadata can reduce the NVM lifetime significantly. On the other hand, the lazy update scheme only updates the parent of the dirty evicted security metadata cacheline, and relies on the eviction process to upwardly propagate the updates [19, 99]. The lazy update scheme reduces the number of extra writes significantly; however, it introduces the risk of losing the cached updates in case of a crash, which renders the integrity verification of the NVM to fail as the root does not reflect the most recent state of the NVM. Therefore, systems implementing the lazy-update scheme do not support persistent security.

Persistent Security

Recent studies showed that systems implementing lazy update schemes lack crash consistency [15, 55,64,66,80,90,91,93,99,101]. Therefore, researchers have approached the problem from different angles. SuperMem [101] proposed a write-through counter cache to avoid encryption counters loss in case of crashes. Liu et al. [55] proposed a scheme to ensure the atomicity of encryption counters. Osiris [90] proposed to recover the encryption counters by using a stop-loss mechanism. Anubis [99] looked at the problem of the recovery time after crashes. Below we describe the state of the art schemes to recover encryption counters (Osiris [90]), and reduce the recovery time (Anubis [99]).

Encryption Counter Recovery Scheme (Osiris): Osiris scheme [90] proposed a mechanism to recover the lost encryption counters by leveraging the Error Correction Code (ECC) bits as a sanity check. Osiris persists the encryption counters after each N-th write, calculates the ECC bits over the plaintext and stores it along with the ciphertext. During the recovery phase, Osiris reads the ciphertext and decrypt it using the encryption counter, then calculates the ECC of the decrypted text and compares it with the stored ECC bits. If the calculated ECC matches the stored ECC, it means the used encryption counter is correct, and if a large number of errors are detected, it means a wrong encryption counter was used.

Security Metadata Cache Recovery Scheme (Anubis): the Anubis scheme [99] proposes a mechanism to reduce the recovery time required to build the MT. Anubis has two versions, AGIT and ASIT. AGIT is a mechanism to recover the security metadata cache content when general MT is used. AGIT allocates a designated region in the memory called the shadow region. Whenever a MT node is modified in the security metadata cache, AGIT persists the address of the modified node in the shadow region. To ensure the integrity of the shadow region, AGIT implements a small general MT (3-4) levels over the shadow region, the small general MT is eagerly updated and its root is kept in the processor. During the recovery, the integrity of the shadow region is verified by

rebuilding the small general MT, and the lost cached nodes are fixed by recalculating the hashes of the child nodes. ASIT is the mechanism used to recover the ToC. Rebuilding the ToC tree is not possible due to intermediate nodes inter-dependencies, therefore ASIT persists all the dirty cached ToC nodes into the shadow region. During the recovery, ASIT verifies the integrity of the shadow region and copies the shadow region content to the security metadata cache to restore the system to its pre-crash state.

Motivation

Security metadata coherence might look as a typical data coherence problem, but it is more complicated than that. For instance, data coherence can be achieved using software, which is not possible for security metadata as it's transparent to software. A data cacheline update requires propagating the updated cacheline to the PEs caching it. On the other hand, a single data cacheline update requires updating its associated encryption counter and the whole MT branch, which might be cached by all the PEs in the system. While it is unlikely to have multiple PEs modifying the same encryption counter at the same time, the likelihood of multiple PEs updating the same MT node significantly increases as the node's level increases. Thus, PEs sharing the memory region would still have to communicate the security metadata updates even if they do not share data. Due to the MT structure, communicating the updates would require sending tens of cachelines. Additionally, the security metadata updates must be persisted along with the data in an atomic manner to prevent crash consistency problems. Therefore, sending the updates should be done in an atomic manner as well, which requires locking the updated MT branch and waiting to receive acknowledgments from all other PEs before proceeding. The pending updated security metadata cachelines held in the write buffer can lead to filling the buffer and may cause processor stalls due to the inability to replace security metadata cachelines. Finally, the update process of the MT requires updating the full MT branch which can lead to high contention over the MT nodes, especially in high MT

levels.

For instance, assuming n PEs operating on a shared 1TB memory that is protected by a 12-level MT. In case of the PEs are updating the same data page (same encryption counter cacheline), each PE would have to lock the 12 nodes of the MT branch before updating, then it needs to ensure the update is acknowledged by all other PEs before unlocking the nodes. On the other hand, assuming two PEs where the first PE is updating 1GB of the memory and the second PE is updating an adjacent 1GB of the memory, the two PEs will not share the first eight levels of the MT, but would still need to share the top 4 levels. Clearly, using an invalidation-based coherence scheme (e.g., MESI/MOESI) will keep invalidating upper levels of the MT and cause higher number of memory accesses. On the other hand, using an update-based coherence scheme will avoid the excessive memory reads, but would still have high performance overheads. Moreover, both schemes will generate high number of memory writes to persist the updated MT nodes, which are required for crash consistency support.

Table 5.1: Metadata management schemes comparison.

Solutions	Cache-efficiency	Writes	Performance	Crash-consistency
CCCM-Invalidate	✗	✗	✗	✓
CCCM-Update	✓	✗	✗	✓
Minerva	✓	✓	✓	✓

Table 5.1 shows the characteristics of the coherence schemes. Note that to achieve crash consistency, typical invalidate (MESI) and update schemes need to send tens of writes and lock the updated MT nodes until acknowledged by all other PEs, which translates to higher performance overheads as shown in Figure 5.4. We designed two different Crash Consistent Coherence Messages (CCCM) schemes, the first is based on MESI protocol which we refer to as CCCM-Invalidate, and

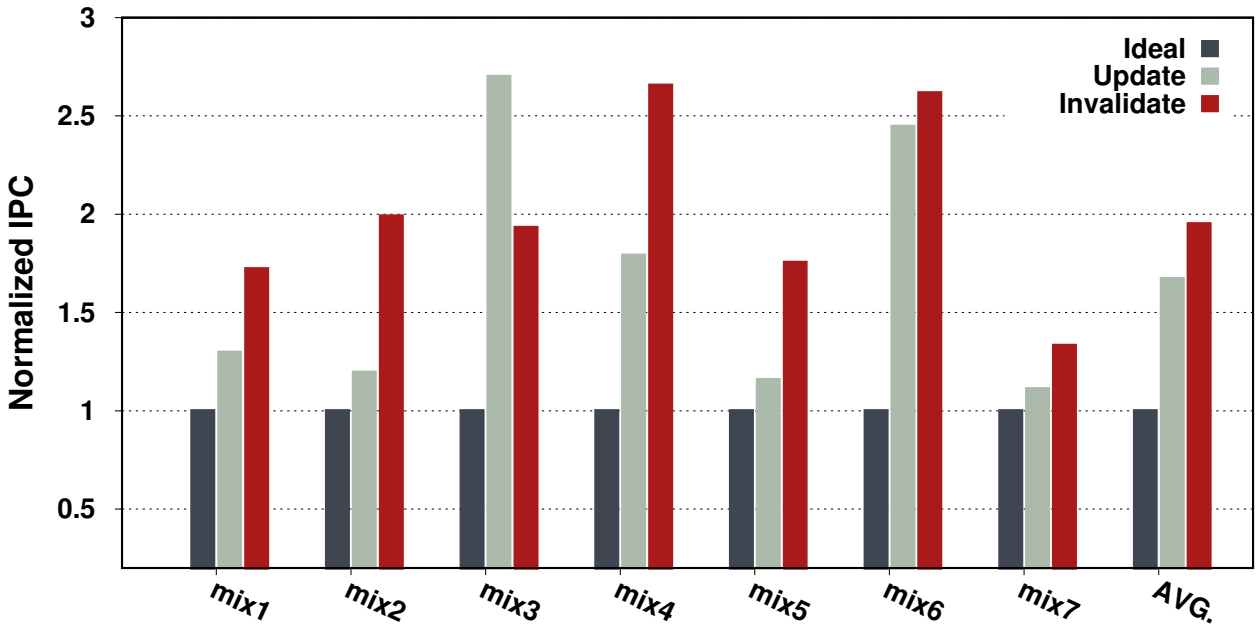

Figure 5.4: The performance overhead of traditional coherence schemes.

the second is based on update protocol which we refer to as CCCM-Update.

Design

In this section, we discuss Minerva's design in light of the threat model discussed in 5, possible design options, and their trade-offs. Before discussing the possible design options, we start with the design requirements.

Security Requirements

Secure memory implementation limits the trust base to the processor chip only, which requires protecting the data integrity and confidentiality when the data leaves the processor chip. In FAM architecture, having multiple PEs in the system introduces several attacks that are not possible in processor-centric architecture. To protect against these attacks, the design should meet the

following requirements:

- **Ensure the Data Integrity:** As each PE is caching a subset of the security metadata, the system should ensure all PEs are able to verify the data's integrity using the most recent version of the MT nodes. Thus, the system should allow PEs to communicate without PE-PE packets dropping.
- **Prevent Encryption Counter Reuse:** As mentioned earlier, EC reuse compromises the counter mode's security, which can happen if multiple PEs are updating the same EC simultaneously.
- **Ensure Correct Updates of the MT:** Having multiple PEs updating the MT simultaneously can lead to incorrect updates of the MT branch if ordering was not followed strictly.

Note that these requirements are hard to achieve in FAM architectures due to having multiple PEs, wherein each PE is caching a subset of the security metadata, where they need to communicate in a timely and securely manner.

Design Requirements

The design should meet the requirements necessary to allow wide adoption and high-performance while ensuring the system's security. In summary, the requirements are as follows:

- **High-Performance:** the design should allow secure memory implementation with minimal performance overhead.
- **Memory Utilization:** increasing the memory utilization requires the design to enable PEs to share the memory, even if the running applications are not sharing the same pages.

- **Scalability:** addressing the coherence problem should be done with minimal number of exchanged messages to allow scalability.
- **Crash Consistency:** ensure a consistent state of the data and its associated security metadata to allow recoverability.
- **NVM Friendly:** ensuring the previous requirements with minimal number of writes to avoid shortening the NVM lifetime.

To put these requirements in the context of FAM architecture, we can imagine a small system of few PEs in a memory centric architecture where each PE is accessing different files. In such scenario, whenever a PE updates a data cacheline, it needs to update the corresponding EC and the whole MT branch. Ideally, each PE should be able to modify its data without notifying other PEs about the update, which provides ***high-performance*** and minimizes the traffic, which provides ***scalability***. However, ensuring the coherence requires notifying the memory sharers by either invalidating or updating the modified cachelines, which can be mitigated if each PE is assigned a separate memory region. Such mitigation technique defies the purpose of FAM architecture, as it disables the effective data sharing and mandates going through a home processor to access other PEs' data. Therefore, an ideal scheme should allow each PE to share the memory where each PE can modify its data without the need to notify other PEs, which provides ***memory utilization***. Finally, imagine a scenario where all the PEs are running and one of the PEs crashed. The crash can render the integrity of the whole shared memory region to be unverifiable, which affects all other running PEs. Therefore, the system should ensure the data and its security metadata are consistent, which provides ***crash consistency***. However, providing crash consistency requires the security metadata to be persisted atomically along with the data. While such scheme can ensure the system's recoverability, it exacerbates the NVMs write endurance and severely reduces its lifetime. Thus, an ideal scheme should allow recoverability while maintaining a low number of

writes, which provides a ***NVM friendly*** design.

Potential Design Options

Now we discuss several straightforward design options which can potentially meet the design and security requirements.

Option 1: Centralized security metadata handling to avoid the problem, achieved by having one PE handling all the encryption and integrity verification operations. While such scheme can eliminate the security metadata coherence problem, it can create a single point of failure at the trusted PE and increase the performance overhead due to throttling the trusted PE. Additionally, this technique defies the purpose of FAM, where PEs can access the shared memory directly. Similarly, the security metadata handling can be done using in/near memory computing techniques, while using a point-point encryption to protect the data in transient as proposed by Awad et.al., [21]. However, such scheme does not limit the trust base to the PEs only, but expands it to include the logic in the memory modules.

Option 2: Allocate specific memory region per PE, where each memory region is protected with different ECs and a separate MT. While this scheme will eliminate the security metadata coherence problem, such a scheme have several drawbacks. First, it prevents the PEs from effectively sharing the memory and reduces the memory utilization, which defies the purpose of FAM. Second, accessing a memory region that belongs to another PE requires going through the owner PE, or moving the data from the owner's memory region to the requester's memory region, which requires re-encrypting the data using the requester's metadata. Alternatively, the owner can share its encryption key with the requester to allow the access to the owner's region, which would require coherence between the two. Third, such scheme requires complex key management scheme to handle the credentials used by each PE.

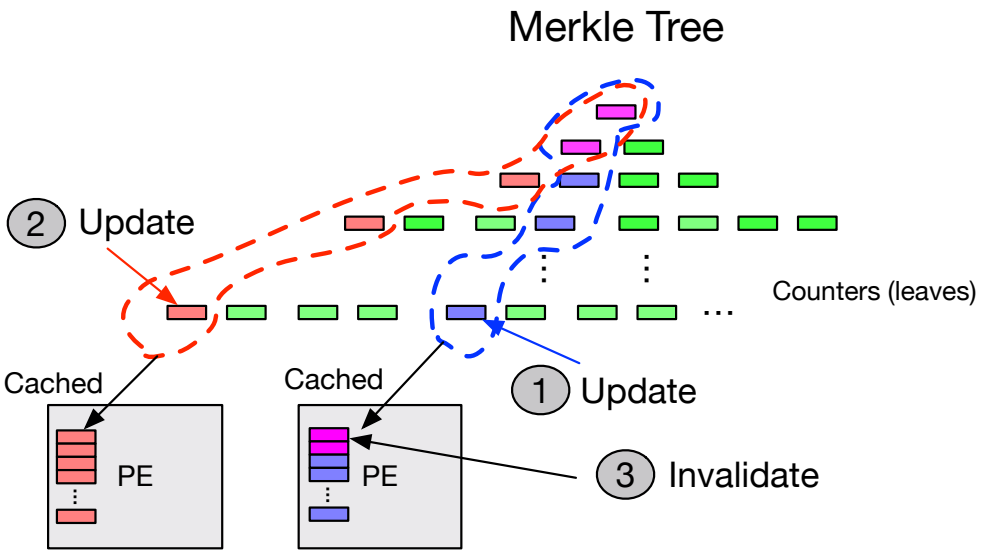

Figure 5.5: Update process in invalidation-based security metadata coherence protocols.

Option 3: Utilize existing coherence mechanisms used for data coherence. While data coherence can be achieved using software or hardware techniques, security metadata is transparent to software which limits the possible solutions to hardware techniques only. Hardware coherence mechanisms can be classified into invalidate or update based schemes. While the invalidate-based schemes i.e., MESI/MOESI are more popular in data coherence, due to the reduced traffic, using such schemes can lead to higher overheads in case of security metadata, due to frequently invalidating shared metadata cachelines. Figure 5.5 illustrates the effect of having multiple PEs updating different data files in a shared memory region. Note that despite the PEs are updating different files, they still affect each other, due to sharing the same MT as discussed in section 5.

While the first design option can meet the security requirements, is NVM friendly, crash consistent, and provide good memory utilization, it lacks the scalability and will have a high performance overhead. On the other hand, the second option can meet the security requirements, is NVM friendly, scalable, crash consistent, and have a low performance overhead, but it does not allow the memory sharing and thus reduces the memory utilization. Finally, the last design option supports memory

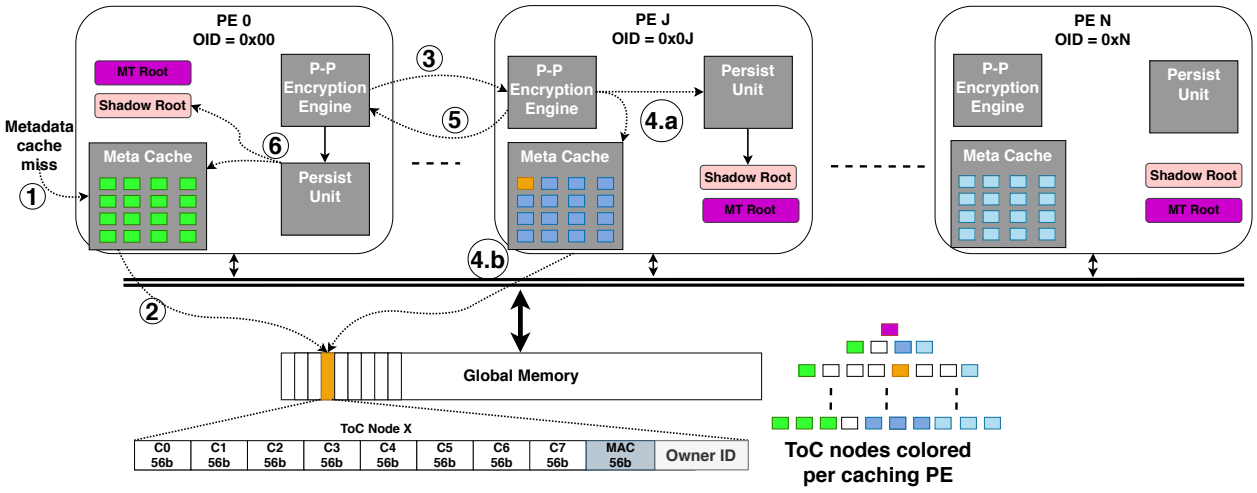

Figure 5.6: Minerva's design.

utilization and crash consistency, but lacks the scalability, not NVM friendly, and have a high performance overhead as shown in Figure 5.4. Additionally, it requires using a universal system clock and messages integrity protection to meet the security requirements.

Minerva's Overview

Before delving into the details of Minerva, we discuss how Minerva meets the design requirements, and maintains the system's security against the attacks mentioned in Section 5.

To maintain the system's security, Minerva uses counter-mode encryption to ensure the confidentiality, and implements a Tree of Counters (ToC) to ensure the integrity. While these measure can protect the data at rest, dropping PE-PE data packets can lead to ECs reuse and integrity verification failures. Thus, Minerva implements PE-PE encryption as discussed in previous work [72]. Combining these techniques Minerva ensures the system's security as discussed later in Section 5.

Minerva's design implements a novel lazy-invalidation scheme that reduces the number of updated MT nodes for each data cacheline update. Furthermore, the lazy-invalidation scheme makes the

security metadata coherence messages a function of the security metadata cache insertion time instead of the update time. Additionally, Minerva enforces an exclusive security metadata caching policy, in which a security metadata cacheline can be cached by one PE at most at any moment. Finally, Minerva implements a cheap directory-like scheme by utilizing the unused 8-bits in each ToC node to store the owner-ID of each security metadata cacheline. As figure 5.6 shows, whenever a PE suffers a security metadata cache miss, it starts by reading the requested node from the memory to obtain the owner-ID as in steps 1 and 2. In Step 3, the requester sends a request to the owner, which updates the requester's cacheline owner-ID, invalidates the requested cacheline from its cache and sends it to the requester in steps 4.a, 4.b, and 5. Finally, the requester updates its persist unit and insert the requested cacheline into the security metadata cache in Step 6.

Note that the exclusive caching allows the owner PE to modify its cached nodes without the need to notify other PEs. Additionally, the lazy-invalidate scheme limits the security metadata updates to the ECs only, and only updates the parent when the EC is evicted. Finally, as the PEs are trusted, Minerva does not need to update the node's parent when the node is transferred from one PE to another, as it would be enough to update the persist unit of the requester to ensure crash consistency.

Minerva's Design

The performance overhead of maintaining security metadata coherence is stemming from, ① the number of processing elements caching the updated cachelines, and ② the number of updated cachelines for each memory write. To address the first problem, Minerva enforces exclusive security metadata caching, which allows each PE to operate on its cached security metadata without sending coherence messages to other PEs. However, despite the exclusive caching, Minerva allows the PEs to efficiently exchange the cached security metadata as we discuss later. To address the

second problem, using a lazy-update scheme may look as it can reduce the number of updated MT nodes. However, an eviction of a dirty encryption counter can trigger a chain of updates, in which each upper node's owner would have to fetch its parents to update them before transferring its owned node. Thus, Minerva implements a novel lazy-invalidate scheme, which distinguishes between the security metadata cache evictions, and transferring the node to another PE in the system. In the contrary of the node's level lazy-update scheme, Minerva's lazy-invalidate scheme is a system's level scheme, which considers the security metadata caches of all PEs as a single unit. Thus, whenever a security metadata cacheline is sent/received from another PE, the lazy-invalidate scheme does not require updating the transferred node's parent.

So far, Minerva's design points can improve the performance at the node level, by eliminating the need to communicate the updates to other PEs. However, on a security metadata cache miss, the PE needs to broadcast the request to obtain the node from the memory, or the PE that is currently caching it. As the security metadata caches are expected to have a high hit rate, the broadcasts are expected to be minimal. However, minimizing the frequency of the updates is essential to improve the scalability. which can be realized if we have a way to identify if/where each node is cached.

Node Ownership Tracking

Minerva employs a directory-like scheme by maintaining the owner-ID of each cached security metadata node in the memory. Note that providing security measures for the region containing the owner-IDs will result in broadcasting the updates each time a node's owner-ID is updated. Thereby, Minerva uses a dedicated unprotected memory region For general MT. On the other hand, each ToC node contains an unused 8-bit which can be used to tag the node with the current caching PE, therefore we will be focusing on the ToC but the same thing applies for general MT. The unused bits can support 255 PEs and the memory as (0x00). Thus, whenever a security

metadata cache miss happens, the requesting PE reads the node's owner-ID tag from the memory, then sends a read-invalidate request to the owner. On receiving the request, the owner writes the requester's ID as the node owner-ID, then sends the requested node. Keeping in mind, the transfer operation is done as a transaction to prevent data races. Using the owner-ID tag will reduce the communication whenever a PE requests a node which is cached at another PE, but for nodes that are memory tagged (0x00), the request has to be broadcasted to prevent duplicate node's caching. As mentioned earlier, owner-ID tag bits are not protected, therefore an attacker can tamper with these bits. However, tampering with these bits will result in requesting a node from a PE that does not have it in its cache, which leads to a broadcast. Therefore, tampering will be detected and it will only lead to a broadcast. On the other hand, protecting the owner-ID bits will cause more overhead than broadcasting. Even though tagging each node with the owner-ID can eliminate the broadcasts for nodes cached by other PEs, broadcasting the request of a memory owned node, and the root each time it is updated would still be required to prevent replay attacks. Note that Minerva's exclusive caching reduces the traffic and removes the need to notify other PEs when a security metadata cacheline is updated. Despite the advantages of such technique, it has a corner case where it can result in a ping-pong behavior. Having two or more PEs actively updating the same data page (protected by the same EC cacheline), then evicting the data from the caches and keeping its encryption counter cached, then reading the cacheline again after being updated and written to the memory by another PE. However, even in such case, constantly transferring the node would still have better performance than updating/invalidating the cacheline frequently. Note that this case will not happen if the requested data cacheline is still cached in another PE, as the cached data's integrity was verified when it was cached earlier.

After minimizing the broadcasts, we discuss how to securely exchange the security metadata nodes between the PEs.

Secure Messaging

As our goal in this work is to provide a coherent view of the security metadata in FAM architecture, protecting exchanged messages between PEs is a must. Rogers et al. [72] showed that protecting processor-processor communication can be easily achieved by utilizing the already existing AES engine. The proposed scheme encrypts the communicated messages and sends a MAC along with the message. While the encryption protects the confidentiality of the sent messages, the MAC is used to prevent reply attacks [72].

After clarifying how to improve each node's performance, minimize the broadcasts, and protect the PE-PE communication, we now discuss the main enabler of these optimizations.

Distributed Crash Consistency and Recovery

Minerva has multiple PEs transferring MT nodes to each other upon request. Therefore, Minerva employs a persist unit similar to Phoenix [15] for each PE, but changes the persist unit to include the MT nodes received dirty from other PEs in the system.

During the recovery phase, each PE uses the shadow region of the persist unit to restore its security metadata cache to its pre-crash state. Note that, Phoenix keeps a persisted copy (shadow) of the dirty cached ToC nodes and restores the system by copying the shadow back to the cache. Therefore, each PE can successfully restore its security metadata cache. However, the owner-ID bits are not protected and an attacker might tamper with these bits during the crash. Even though tampering with these bits is not going to compromise the system's security, but it will lead to unnecessary broadcasts, therefore, each PE updates the owner-ID bits of its cached metadata during the recovery. During the recovery stage, one of the PEs orchestrates the recovery process (assuming PE0) to prevent memory throttling. The recovery manager (PE0) starts by recovering its cache

and once done, it sends an initiate recovery signal to PE1, once PE1 finishes the recovery, it sends a recovery done signal back to PE0, and so on. If any of the PEs fails to recover successfully, a recovery failure signal is raised and tampering is detected. Once all PEs finish the recovery, PE0 sends a recovery done signal to all PEs and processing may resume.

Design Discussion

In this Section, we discuss some of Minerva's design options. Minerva was designed based on our understanding and expectations for future FAM architectures, but in case our vision was different than reality of FAM architectures, Minerva would still be applicable.

Minerva scheme addresses the problem of security metadata coherence in small scale FAM architectures where tens of nodes are sharing the memory region. However, Minerva can be applied if the memory is divided into regions where each region is shared by small number of nodes, and each region has its own Merkle tree. We leave systems with huge number of nodes sharing the same memory region for future work. Minerva assumes FAM architectures to employ NVM as the shared memory. While memory encryption and integrity verification would still be required for DRAM, providing crash consistency will not be required. Therefore, the persist units will no longer be required if DRAM were to be used as the shared memory. Moreover, Minerva uses a ToC for integrity verification but using a general MT is also possible. However, node ownership will need to be kept in a separate table in the memory, as the general MT nodes are hashes of their lower levels. While using a general MT will only affect the nodes' ownership, using an eager update is very costly and will have serious scalability issues due to the root broadcasts. Finally, Minerva limits the trust boundary to the PEs only, as in state-of-the-art schemes [19, 55, 90, 99, 101]. However, if the connecting fabric (Gen-Z) were to be trusted, there will be no need for the point-point encryption.

Security Discussion

To ensure the system's overall security, we need to protect the data confidentiality and integrity at rest and while being transferred.

Minerva protects the data at rest using the counter mode encryption and ToC. Thus, an attacker cannot modify the data without being detected. However, the owner-ID bits are not protected. Thus, an attacker can tamper with these bits, but such an attack can lead to requesting an MT node from a PE that does not have it, which leads to a broadcast to retrieve the most recent node and detect the attack. To ensure data integrity at rest, the system should enable the PEs to securely communicate and obtain the most recent MT nodes.

Minerva allows the PEs to securely communicate by using a PE-PE encryption and integrity verification engine as described in Section 5. By protecting the PE-PE communication, Minerva ***ensures data integrity*** as it enables the PEs to obtain and verify the integrity of the most recent security metadata. Additionally, by employing the exclusive caching, Minerva ***prevents EC reuse*** as the EC can be cached by one PE, which can be securely transferred to other PEs. Finally, Minerva ***ensures correct MT updates*** by applying the lazy-invalidate scheme, which does not require to update the whole MT branch. Thus, Minerva does not require a universal system clock to ensure the correctness of MT updates.

Methodology

We modeled Minerva, CCCM-Update and CCCM-Invalidate security metadata coherence schemes in the Structural Simulation Toolkit (SST) simulator [70]. To accurately model our work, we modified the memory controller to handle the security metadata, we implemented the coherence messaging exchange scheme, the security metadata caches, and the message communication between

PEs. We assume an encryption latency of 24-cycle as in [90]. The configurations of the simulated PEs and the memory system are shown in Table 5.2.

Table 5.2: Configurations of the simulated system.

System Configuration	
Number of PEs	2, 4, 8, 16
Fabric latency	40ns
Fabric Attached Memory	
Technology	PCM-based NVM
Capacity	16GB
Latencies	Read 60ns, Write 150ns
Processing Element (PE)	
Processor	4 Cores, x86-64, Out-of-Order, 2.00GHz
L1 Cache	Private, 4 Cycles, 32KB,8-Way
L2 Cache	Private, 6 Cycles, 256KB, 8-Way
L3 Cache	Shared, 12 Cycles, 1MB/core, 16-Way
Cacheline Size	64Byte
Encryption Parameters	
Security Metadata Cache	256KB, 8-Way, 64B Block

Table 5.3: Workloads.

Workload	Benchmarks
Mix1	(N/2) arswp, (N/2) btree
Mix2	(N/2) Hashmap, (N/2) randwr
Mix3	(N/2) randwr, (N/2) rbtree
Mix4	(N/2) seqwr, (N/2) arswp
Mix5	(N/2) seqwr, (N/2) hashmap
Mix6	(N/2) tpcc, (N/2) btree
Mix7	(N/2) tpcc, (N/2) rbtree

To evaluate our proposed scheme, we ran combinations of the persistent applications shown in Table 5.3, similar to previous work [56]. Each PE is running a different benchmark for 500M instructions. In our model, we connected all the PEs to share the fabric attached memory, where each PE has a cache hierarchy as described in Table 5.2. The functionality of these applications are shown in Table 5.4. Additionally, these persistent applications expect to recover from crashes and thus the application's data needs to be NVM resident, which requires data writes to be flushed

Table 5.4: Benchmarks description.

Benchmark	Description	MPKI
ARSWP	Swap random elements of an array	4.45
RANDRW	Random updates to persistent memory	3.25
SEQRW	Sequential updates to persistent memory	27.41
BTREE	Insert and look up random elements in b-tree	3.14
RBTREE	Insert and look up random elements in red-black tree	16.42
TPCC	N-Store variant to measures the performance of online transactions	4.63
HASHMAP	Hashmap implemented with NVML [83] library	41.15

all the way to the NVM, and consequently, the security metadata of the NVM needs to be updated with each write.

Evaluation

In this Section, we discuss the evaluation results of the CCCM and Minerva schemes against an ideal coherency scheme that incurs no overheads.

Minerva's Impact on ToC Accesses

Minerva implements a lazy-invalidate scheme that reduces the accesses to upper MT levels, which results in less frequent communication of MT updates. Figure 5.7 shows Minerva's accesses to the ToC levels comparing to other schemes. We observe that Minerva shows a similar behaviour as the ideal scheme, but generating 18% more accesses to the ToC than the ideal scheme. As shown in the figure, Minerva serves 65.8% of the requests from the first ToC level, 10% from each level of

levels 2-4, 3.5% from level5, and less than 0.3% from upper levels. Thus, Minerva only requires to request for a security metadata cacheline's ownership on a very small percentage of requests.

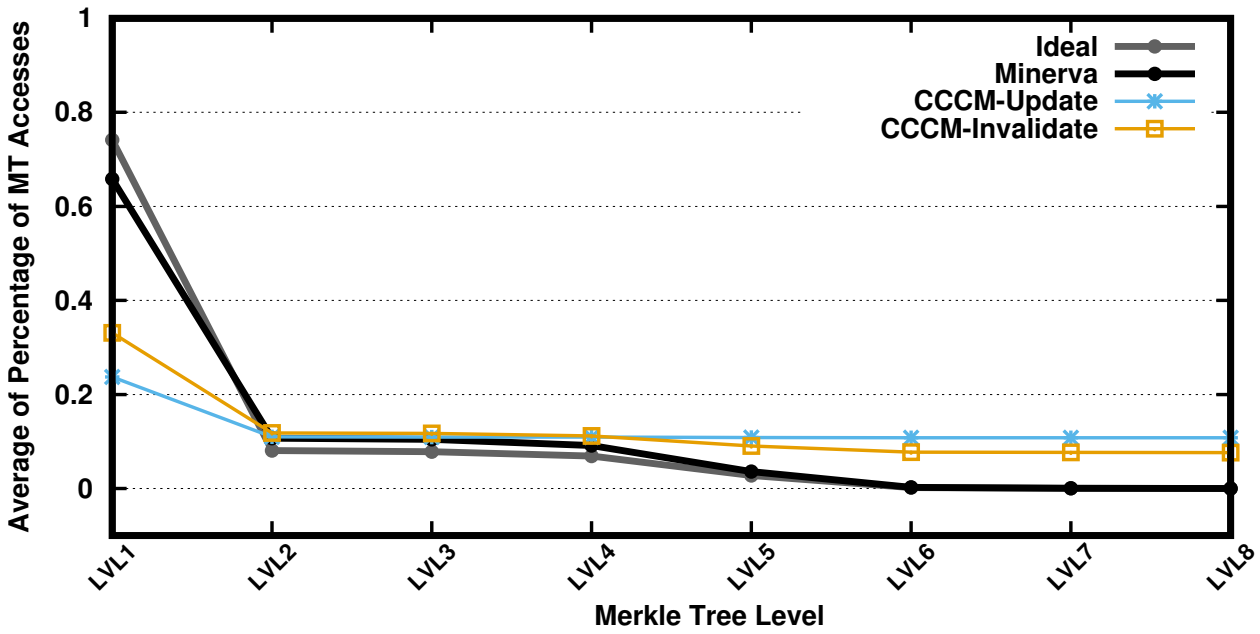

Figure 5.7: Percentage of accesses to different MT levels.

On the other hand, the CCCM schemes generate 3.16x and 2.21x for the invalidate and update schemes respectively. We observe that majority of the ToC requests in the update scheme are caused by updating all the ToC levels, and the invalidate scheme requests are generated because of invalidating the nodes of upper ToC levels. As shown in Figure 5.7, the update scheme is serving 23.7%, and the invalidate scheme is serving 33% from the first level. Upper levels are serving 11.8%-7.7% each. Therefore, these updates are causing the high overheads of the CCCM schemes.

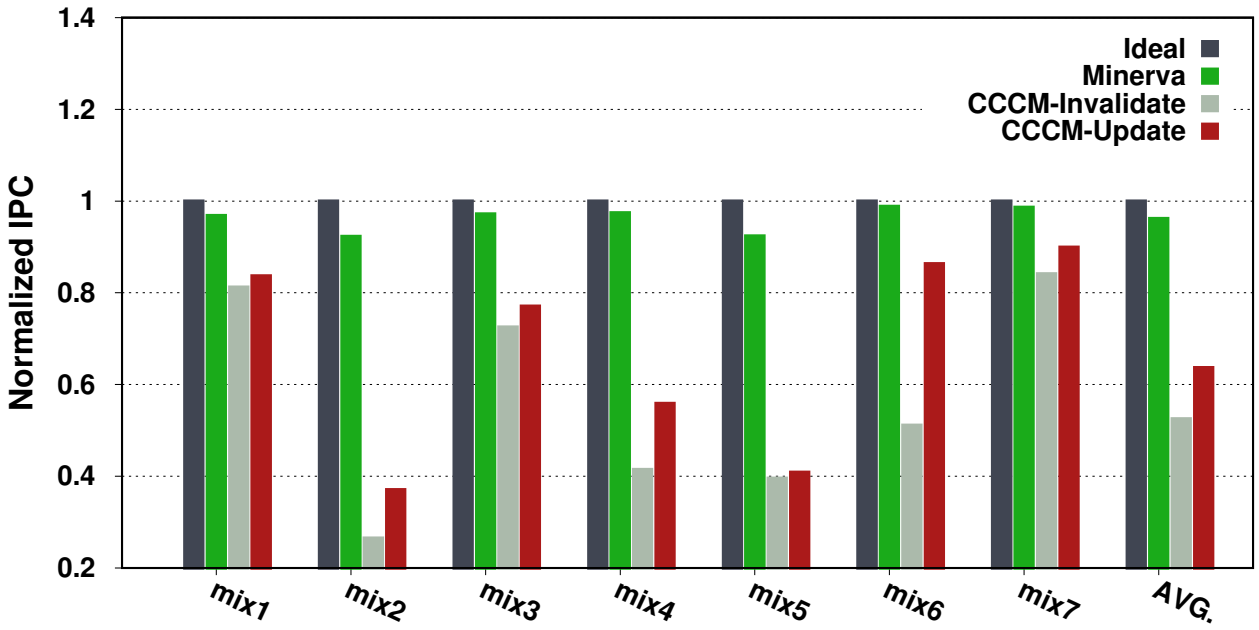

Figure 5.8: Minerva's impact on performance.

Minerva's Impact on Performance

As shown in Figure 5.8, Minerva has a very low performance overhead as it has an average normalized IPC of 96.2%, CCCM-Update has 63.6%, and CCCM-Invalidate has 52.5%. The performance overhead of the CCCM schemes is a function of data cache's miss rate (dirty data evictions). For instance, Mix2 and Mix5 have the lowest data cache's hit rates of 87%. However, Mix5 is performing better as half of the PEs are executing *seqwr* benchmark, which has a low MPKI that reduces the frequency of sending coherence messages to invalidate/update security metadata cachelines in other PEs. The performance of other Mixes are a direct function of the data cache's hit rate. We observe that CCCM-Update scheme is always performing better than CCCM-Invalidate scheme, which is caused by the frequent invalidation of security metadata cachelines that results in memory reads.

On the other hand, Minerva's performance is a function of the security metadata cache's miss rate. Thus, workloads with the highest security metadata cache's miss rates are incuring the highest per-

formance overhead. As shown in Figure 5.8, Mix2 and Mix5 are incuring the highest performance overhead in Minerva, which is caused by the low security metadata cache's hit rate. Note that Minerva's performance overhead is minimal as the security metadata misses are actually a fraction of the data caches' misses, with an average data caches' hit rate of 95% and an average security metadata cache's hit rate of 67.1%, Minerva only needs to request security metadata cachelines from other PEs on 1.6% of the memory accesses.

Minerva's Impact on Number of Writes

Figure 5.9 shows the normalized number of memory writes incurred by Minerva. The number of writes for CCCM-Update and the CCCM-Invalidation schemes are not shown as it is **8x**, due to the need to persist the whole ToC path on each update.

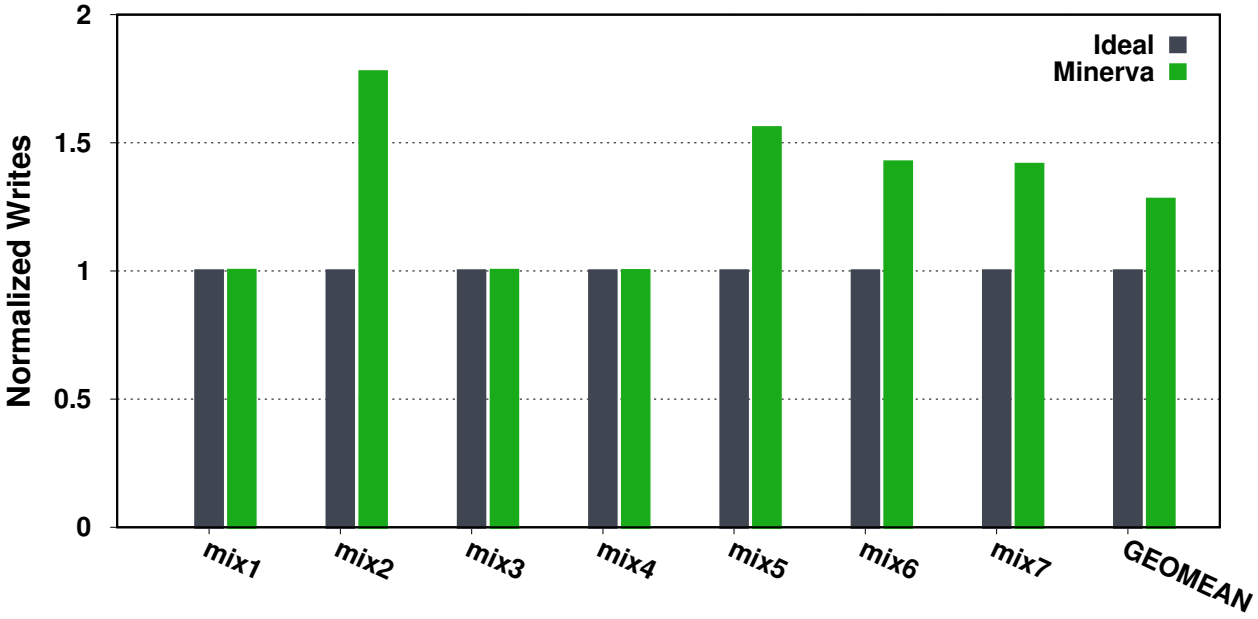

Figure 5.9: Minerva's impact on the number of writes.

In comparison, Minerva increases the average number of shared memory writes only by 27.9%. Minerva updates the ToC node's owner-ID in the memory for each security metadata cache's miss.

Therefore, workloads with high security metadata miss rates are showing high overheads. However, Mix6 and Mix7 are showing a high increase in the number of writes, which is caused by *TPCC* benchmark that is a read intensive and has a low number of writes. Therefore, the owner-ID updates are showing a high increase in the writes overhead. We Observe that Mix2 and Mix5 are showing the highest increase in the number of writes, which is caused by the frequent owner-ID updates. Note that Minerva's write overhead is a direct function of the security metadata cache's hit rate, therefore, workloads with the lowest security metadata hit rates will incur the highest increase in the number of writes. We observe that Mix5 is showing a higher increase in the writes, which is caused by *randwr* benchmark that has a higher MPKI than *seqwr* used in Mix5.

Minerva's Impact on Number of Reads

Figure 5.10 shows the normalized number of memory reads for the evaluated schemes. CCCM-Update has an average read overhead of 0.26% caused by receiving and caching security metadata from other PEs, which causes few evictions of required metadata. The CCCM-Invalidation scheme has an average of 519.9% number of reads, which is caused by invalidating upper ToC levels. On the other hand, Minerva has an average reads overhead of 14.7% which is caused by reading the owner-ID of the requested node before sending the request to the owner. Mix2 and Mix5 are showing the highest increase in the number of reads, which is caused by the low security metadata cache's hit rate for these mixes.

Minerva's Impact on Traffic

Figure 5.11 shows the exchanged packets in the system in different schemes. The CCCM-Invalidation scheme has an average of 413.6% extra packets, caused by the metadata reads, metadata writes, invalidation messages and acknowledgments. The CCCM-Update scheme is causing an average

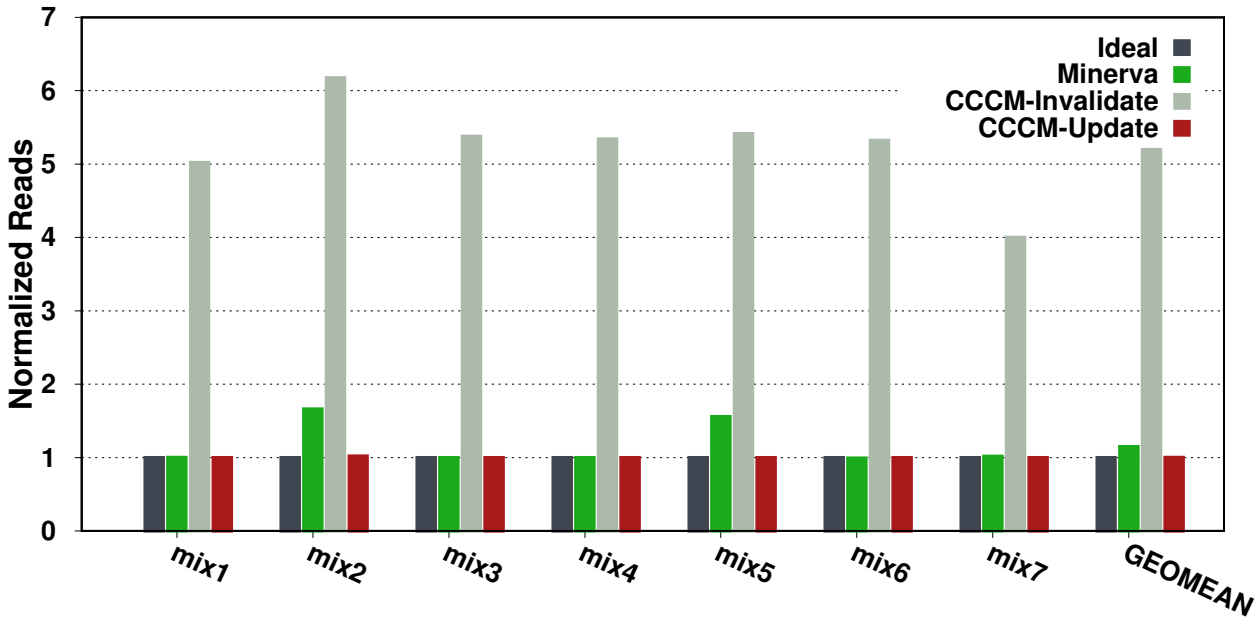

Figure 5.10: Minerva's impact on the number of reads.

of 446.1% extra packets caused by the metadata writes, update messages, and acknowledgments. Note that the CCCM-Invalidation scheme only requires to send the updated encryption counter

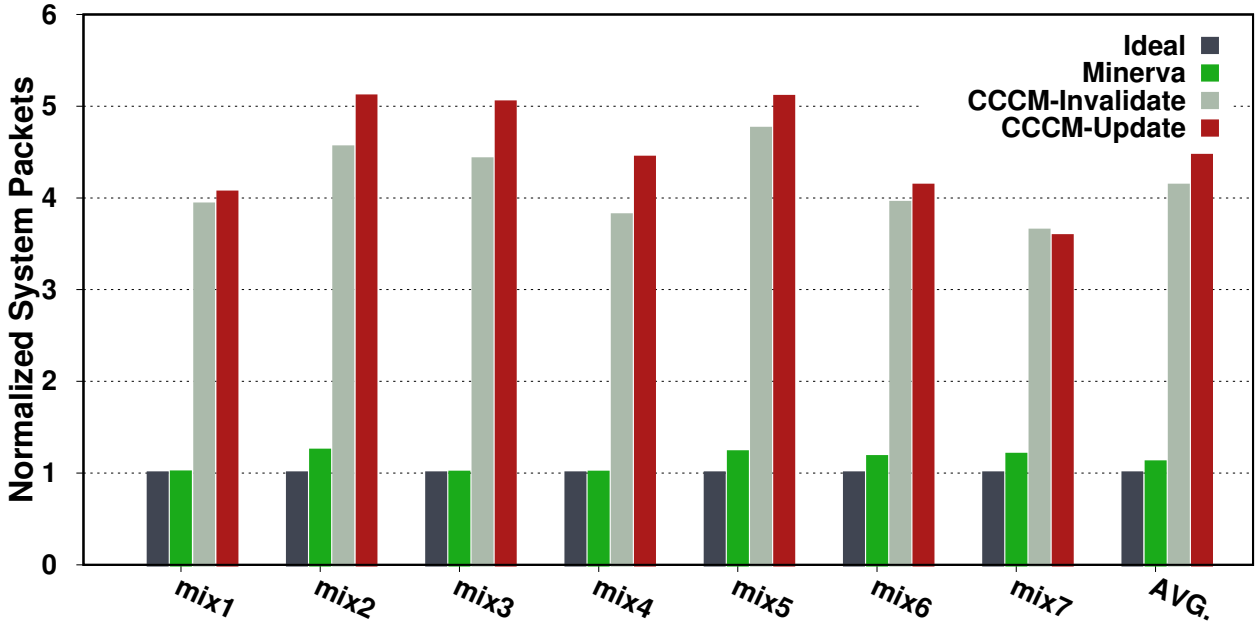

Figure 5.11: Minerva's impact on traffic.

to invalidate the whole branch, while CCCM-Update would require sending the whole branch to prevent multiple reads by each PE to update its cached security metadata nodes, both schemes receive a single acknowledgment message. Minerva is causing an average of 12.1% extra packets caused by the broadcasts to obtain the memory owned nodes, as most of the misses are coming from encryption counters that are not used by any other PE as the applications are not sharing data.

Sensitivity Analysis on Scalability

We analyze how Minerva performs with increased number of PEs in the system (from 2 to 16), and our experimental results show that Minerva barely affects the performance as shown in Figure 5.12. Each performance metric is normalized to ideal memory encryption and integrity verification scheme with the same number of PEs, that assumes coherence without sending any coherence messages.

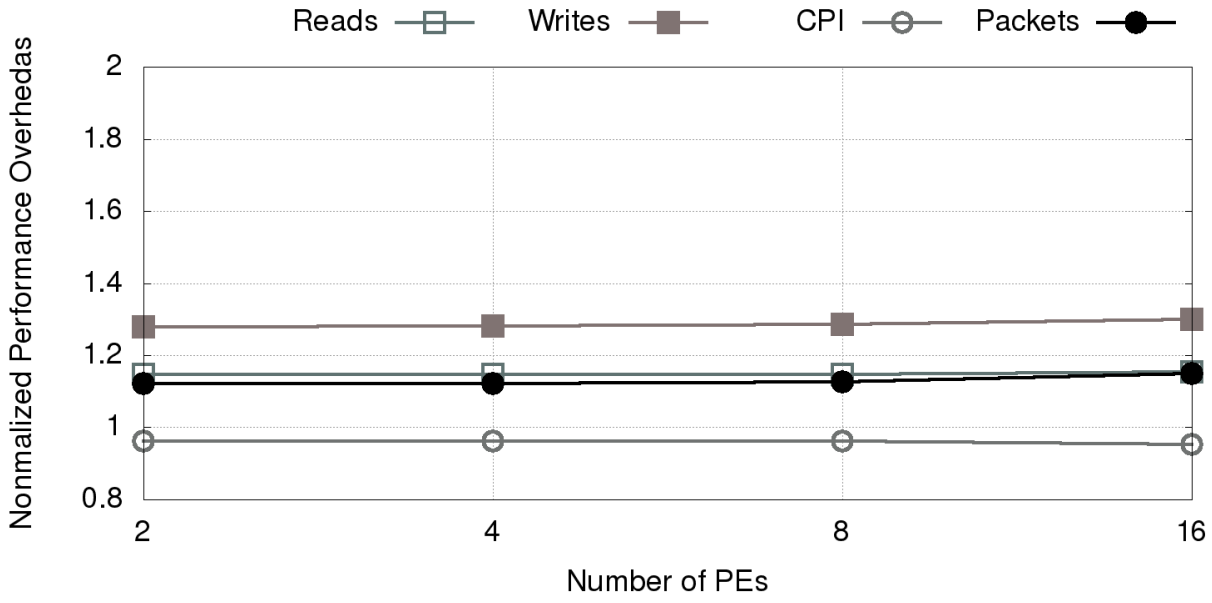

Figure 5.12: Minerva sensitivity to increase number of PEs.

We observe that Minerva's performance stays the same when 4-8 PEs are used, but increases by 1.1% when 16 PEs are used. This small increase is caused by the increase in the broadcast range. The number of memory reads increased by 0.6% when 16 PEs are used, but stays the same for less than 16 PEs. The number of transferred packets increased by 0.4% when scaled to 8 PEs, and increased by 2.3% for 16 PEs due to the broadcasts for memory owned nodes. Note that increasing the number of PEs has two contradicting effects on Minerva's performance. One one side, increasing the number of PEs increases the broadcast range when the root is updated or a memory owned node is requested. On the other side, increasing the number of PEs increases the percentage of PEs owned security metadata, which reduces the number of broadcasts. The writes overhead stays the same for 4-8 PEs, but increased slightly by 1.2% for 16 PEs, which is caused by the owner-ID updates. These result are expected, as Minerva enables secure FAM architectures by adding a single message to the security metadata node's owner, followed by a single write to update the ToC node owner.

Related work

The most related works to Minerva are secure memory in Non-Uniform Memory Access (NUMA) architecture [73], Anubis [99], Phoenix [15], Triad-NVM [19], and Osiris [90]. The proposed scheme in [73] addresses the security requirements for distributed shared memory systems in NUMA architectures. In the proposed solution, each processor is handling its memory security and the security metadata of one processor memory can not be cached by other processors, whenever any node tries to access another nodes' memory it has to go through the home processor for that memory. Anubis [99] scheme targets the recovery time problem of encrypted and integrity protected NVM. Anubis allocates a memory region in the NVM to persist the cache updates, and then uses the allocated region to recover the cache content after a crash. Phoenix [15] is an opti-

mized version of Anubis [99] which reduces the number of writes to the NVM by adding more to the recovery time. Triad-NVM [19] discusses the performance and recovery time trade-off. Triad-NVM suggests persisting N-levels of the BMT to reduce the recovery time. However, Triad-NVM does not work with ToC. Osiris [90] scheme targets the encryption counters recovery problem. Osiris implements a stop-loss mechanism to persist the encryption counters on each N-th write, and rely on the ECC bits as a sanity check to recover the lost counters. Memory encryption engine [34] describes the details of the memory encryption engine of Intel's SGX, and describes the details of the ToC.

Several works have been done in the NVM security and persistency field [28, 45, 68, 76, 77, 81, 96, 100]. Morphable Counters [76] proposes a scheme to pack more encryption counters in a cacheline, thus increases the cache-ability and boosts the performance. Vault [81] proposes a variable arity integrity tree to reduce the tree size and improve cache-ability. However, the majority of the works discuss the performance overhead of NVM encryption and integrity verification, and propose optimizations to reduce these overheads. To the best of our knowledge, we are the first to discuss the problem of security metadata coherency in FAM architectures.

Conclusion

Minerva is a novel memory controller design that solves the security metadata coherency problem in small to medium scale FAM architectures. Unlike traditional coherence mechanisms that require updating the whole MT branch, Minerva uses a lazy-invalidate scheme that limits the update to the encryption counter. Additionally, Minerva reduces the performance overhead significantly due to the exclusive caching, which makes the coherence messages a function of the security metadata cache misses instead of security metadata cache updates. Finally, Minerva ensures the correct updates of the MT without the need of a universal clock, which is required to guarantee the MT up-

dates ordering if traditional coherence mechanisms were to be used. In summary, Minerva achieves security metadata coherence with performance overhead of 5.1% for systems with up to 16 PEs, and reduces the number of extra writes to the NVM to 27.9%.

CHAPTER 6: CONCLUSION

In this dissertation, we addressed several problems when the emerging NVMs are integrated in the system as a sole main memory, part of the main memory, or as a new tier in the memory system. In case of using the NVM as the sole main memory, we addressed the number of extra writes required to enable the system's recoverability for integrity protected NVMs using the tree of counters. To address this problem, we proposed Phoenix. When the NVM is used as a part of the main memory as in hybrid memory systems, we addressed the problem of the performance overheads incurred by persistent applications. To address the problem, we proposed Stealth-Persist. Finally, when the NVM is integrated as a new memory tier as in Fabric-Attached memory architectures, we addressed the performance issues caused by secure memory implementation and the cacheability of security metadata, and the security metadata coherence problem. To address these problems, we proposed caching techniques for security metadata, and Minerva. The contributions of our work are summarized below.

First, when the NVM is integrated as the sole main memory, we proposed Phoenix. Phoenix is based on four observations, ① most updates of the lazily updated ToC are done to leaf nodes.② leaf nodes are the least likely to be evicted as they will be reused frequently for verification and update purposes. ③ leaf nodes can be recovered using any encryption counter recovery scheme, we used Osiris in our work, but any other scheme should work. ④ cached intermediate nodes can be persisted at their location instead of being copied to the shadow region, and the small MT only needs to cover the dirty cached intermediate nodes and the dirty encryption counters. Phoenix achieves recoverability with ultra-low recovery time while keeping the number of writes to the minimum in ToC integrity protected NVMs. Our solution achieves a significant improvement in the number of writes as it reduces the number of writes by 90.8% less than state-of-the-art scheme Anubis, and 3.8% less than the write back scheme, with a recovery time of less than a second in

ToC integrity protected systems. In addition, Phoenix recovery time and extra writes are a function of the cache size, as it works by recovering the lost cached ToC nodes. In summary, Phoenix recovers the ToC in less than a second, reduces the number of writes significantly, and improves the performance.

Second, when the NVM is integrated as a part of the main memory as in DRAM-NVM hybrid memory system, we proposed Stealth-Persist. Stealth-Persist is a novel memory controller design that allows caching the NVM resident pages in the DRAM while ensuring the pages persistency. By serving NVM requests from DRAM, Stealth-Persist exploits bank level parallelism which reduces the memory contention and brings in additional performance gains. Stealth-Persist improves the system's performance of persistent applications in hybrid memory systems by 42.02% on average with Stealth-Persist FTP. However, Stealth-Persist FTP requires significant number of pages to be copied from the NVM to DRAM. With Stealth-Persist MQ approach, we show a performance improvements of 30.09% with reasonable page mirrors. Stealth-Persist achieves this improvement at the cost of a small hardware managed table, a small cache in the memory controller, and by utilizing the WPQ. Further, we show sensitivity analysis by varying the mirroring region size and mirroring threshold level. With a mirroring region size is 8MB, Stealth-Persist MQ-FTP achieves performance improvement of 42.93%.

Third, when the NVM is integrated as a new memory tier as in the fabric-attached memory systems, we proposed using split-tree to ensure the data integrity, then proposed some caching techniques to improve the performance even further. In caching techniques we argue that protecting the confidentiality and integrity of the data in FAM architecture is challenging and requires special handling due to having two different memories. Implementing secure memory architecture schemes directly can introduce higher overheads. Split-Tree is a scheme that uses a dedicated integrity tree to protect the local memory, and another integrity tree to protect the global memory. Using two different integrity trees can reduce the performance overhead of traditional secure memory implementa-

tion schemes. However, having two different trees will cause a high contention over the security metadata cache and can lead to unnecessary performance overheads and extra memory accesses. To reduce the effect of the contention over the cache resources, we partition the security metadata cache to have static partition for the local MT and another partition for the global MT. Furthermore, we prevent caching higher levels of the local MT due to the use of lazy update for the local MT, and we partition the global MT cache sets between different MT levels dynamically. Using Split-Tree can reduce the performance overhead by 7%, global memory reads by 34%, global writes by 140%. However, this improvement is achieved by increasing the local memory reads by 93%, and local memory writes by 29%. Finally, implementing cache partitioning techniques and allowing DRAM caching of the global MT nodes improved the performance by additional 7%, which is stemming from the high improvement of the security metadata cache hit rate.

Finally, when the NVM is integrated as a new memory tier as in the fabric-attached memory systems, we proposed Minerva. Minerva is a novel memory controller design that solves the security metadata coherency problem in small to medium scale FAM architectures. Unlike traditional coherence mechanisms that require updating the whole MT branch, Minerva uses a lazy-invalidate scheme that limits the update to the encryption counter. Additionally, Minerva reduces the performance overhead significantly due to the exclusive caching, which makes the coherence messages a function of the security metadata cache misses instead of security metadata cache updates. Finally, Minerva ensures the correct updates of the MT without the need of a universal clock, which is required to guarantee the MT updates ordering if traditional coherence mechanisms were to be used. In summary, Minerva achieves security metadata coherence with performance overhead of 5.1% for systems with up to 16 PEs, and reduces the number of extra writes to the NVM to 27.9%. Minerva achieves the security metadata coherence in FAM systems, while maintaining the trust base to the processor only, maintains the security of the system, and detects various types of attacks targeting the system's security.

In summary, in this dissertation, we investigated and proposed several design recommendations to address the problems of NVMs' integration in emerging architectures.

LIST OF REFERENCES

[1] "AMD Memory Encryption," http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, accessed: 2020-02-15.

[2] "CCIX specifications V 1.0," https://www.ccixconsortium.com/library/specification/, accessed: 2019-11-10.

[3] "CXL 1.1 specifications," https://www.computeexpresslink.org/, accessed: 2019-11-10.

[4] "Flash Memory Summit 2018 - Persistent memory DIMMs," https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series.html, accessed: 2020-23-07.

[5] "GenZ specifications kernel description," https://genzconsortium.org/faqs/, accessed: 2019-09-30.

[6] "Intel Optane DC Persistent Memory," https://builders.intel.com/docs/networkbuilders/intel-optane-dc-persistent-memory-telecom-use-case-workloads.pdf, accessed: 2020-24-07.

[7] "Intel Promises Full Memory Encryption," https://hardware.slashdot.org/story/20/02/29/0433242/chasing-amd-intel-promises-full-memory-encryption-in-upcoming-cpus, accessed: 2020-03-03.

[8] "Intel® Optane™ DC Persistent Memory Operating Modes Explained," https://itpeernetwork.intel.com/intel-optane-dc-persistent-memory-operating-modes/#gs.xtmcnw, accessed: 2020-24-02.

[9] "Linux Direct Access of Files (DAX)." [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[10] "NVDIMM - Changes are Here So What's Next," https://www.snia.org/sites/default/files/SSSI/NVDIMM%20-%20Changes%20are%20Here%20So%20What's%20Next%20-%20final.pdf, accessed: 2020-03-29.

[11] "NVDIMM-P," https://software.intel.com/content/www/us/en/develop/articles/enabling-persistent-memory-in-the-storage-performance-development-kit-spdk.html, accessed: 2020-02-02.

[12] "NVM Programming Model v1.2," https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf, accessed: 2020-02-02.

[13] "Persistent Memory Development Kit," https://pmem.io/pmdk/, accessed: 2019-03-27.

[14] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for gpus within heterogeneous memory systems," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 607–618.

[15] M. Alwadi, A. Mohaisen, and A. Awad, "Phoenix: Towards persistently secure, recoverable, and nvm friendly tree of counters," 2019.

[16] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 707–722.

[17] A. Awad, S. Hammond, C. Hughes, A. Rodrigues, S. Hemmert, and R. Hoekstra, "Performance analysis for using non-volatile memory dimms: opportunities and challenges," in *Proceedings of the International Symposium on Memory Systems*, 2017, pp. 411–420.

[18] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 263–276, 2016.

[19] A. Awad, Y. Solihin, L. Njilla, M. Ye, and K. Zubair, "Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 169–180.

[20] A. Awad, S. Suboh, M. Ye, K. Abu Zubair, and M. Al-Wadi, "Persistently-secure processors: Challenges and opportunities for securing non-volatile memories," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019, pp. 610–614.

[21] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "Obfusmem: A low-overhead access obfuscation for trusted memories," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2. ACM, 2017, pp. 107–119.

[22] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, and T. Krishna, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, p. 1, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[23] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "Rewind: Recovery write-ahead system for in-memory non-volatile data-structures," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 497–508, 2015.

[24] S. Chhabra and Y. Solihin, "i-nvmm: a secure non-volatile main memory system with incremental encryption," in *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE, 2011, pp. 177–188.

[25] C. C. Chou, A. Jaleel, and M. K. Qureshi, "Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 1–12.

[26] C. Chou, A. Jaleel, and M. Qureshi, "Batman: Techniques for maximizing system bandwidth of memory systems with stacked-dram," in *Proceedings of the International Symposium on Memory Systems*, 2017, pp. 268–280.

[27] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 105–118, 2011.

[28] ——, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," *ACM Sigplan Notices*, vol. 47, no. 4, pp. 105–118, 2012.

[29] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 133–146.

[30] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[31] D. Das, D. Arteaga, N. Talagala, T. Mathiasen, and J. Lindström, "{NVM} compression—hybrid flash-aware application level compression," in *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads ({INFLOW} 14)*, 2014.

[32] B. K. Debnath, S. Sengupta, and J. Li, "Chunkstash: Speeding up inline storage deduplication using flash memory." in *USENIX annual technical conference*, 2010, pp. 1–16.

[33] C. R. Ferenbaugh, "Pennant: an unstructured mesh mini-app for advanced architecture research," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4555–4572, 2015.

[34] S. Gueron, "A memory encryption engine suitable for general purpose processors," 2016, https://eprint.iacr.org/2016/204.

[35] G. Gunow, J. Tramm, B. Forget, K. Smith, and T. He, "Simplemoc-a performance abstraction for 3d moc," 2015.

[36] R. Hagmann, "Reimplementing the cedar file system using logging and group commit," in *Proceedings of the eleventh ACM Symposium on Operating systems principles*, 1987, pp. 155–162.

[37] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, sep 2006. [Online]. Available: https://doi.org/10.1145/1186736.1186737

[38] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[39] Q. Hu, J. Ren, A. Badam, J. Shu, and T. Moscibroda, "Log-structured non-volatile main memory," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 703–717.

[40] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 427–442, 2016.

[41] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "Heteroos: Os design for heterogeneous memory management in datacenter," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 521–534.

[42] I. Karlin, J. Keasler, and J. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2013.

[43] K. Keeton, "The machine: An architecture for memory-centric computing," in *Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, vol. 10, no. 2768405.2768406, 2015.

[44] A. Kokolis, D. Skarlatos, and J. Torrellas, "Pageseer: Using page walks to trigger page swaps in hybrid memory systems," in *2019 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2019, pp. 596–608.

[45] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 481–493.

[46] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[47] V. R. Kommareddy, S. D. Hammond, C. Hughes, A. Samih, and A. Awad, "Page migration support for disaggregated non-volatile memories," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 417–427.

[48] V. R. Kommareddy, C. Hughes, S. Hammond, and A. Awad, "Investigating fairness in disaggregated non-volatile memories," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2019, pp. 104–110.

[49] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 2–13, 2009.

[50] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE micro*, no. 1, pp. 143–143, 2010.

[51] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, 1996, pp. 84–92.

[52] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao, "Cachededup: In-line deduplication for flash caching," in *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, 2016, pp. 301–314.

[53] Z. Li, R. Zhou, and T. Li, "Exploring high-performance and energy proportional interface for phase change memory systems," *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 210–221, 2013.

[54] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 2012, pp. 1–12.

[55] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 310–323.

[56] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing memory and storage support for non-volatile memory systems," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 143–156.

[57] S. Mandal, G. Kuenning, D. Ok, V. Shastry, P. Shilane, S. Zhen, V. Tarasov, and E. Zadok, "Using hints to improve inline block-layer deduplication," in *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, 2016, pp. 315–322.

[58] A. Memaripour, J. Izraelevitz, and S. Swanson, "Pronto: Easy and fast persistence for volatile data structures," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 789–806.

[59] T. S. Messerges, "Securing the aes finalists against power analysis attacks," in *International Workshop on Fast Software Encryption*. Springer, 2000, pp. 150–164.

[60] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 135–148, 2017.

[61] ——, "An analysis of persistent memory use with whisper," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 135–148, 2017.

[62] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis, "Rthms: A tool for data placement on hybrid memory system," *ACM SIGPLAN Notices*, vol. 52, no. 9, pp. 82–91, 2017.

[63] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen, "Mempod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 433–444.

[64] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling,"

in *2009 42nd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*. IEEE, 2009, pp. 14–23.

[65] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, 2009, pp. 24–33.

[66] J. Rakshit and K. Mohanram, "Assure: Authentication scheme for secure energy efficient non-volatile memories," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[67] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the international conference on Supercomputing*, 2011, pp. 85–95.

[68] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 672–685.

[69] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, pp. 1–32, 2013.

[70] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls *et al.*, "The structural simulation toolkit," *SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.

[71] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 183–196.

[72] B. Rogers, M. Prvulovic, and Y. Solihin, "Efficient data protection for distributed shared memory multiprocessors," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, 2006, pp. 84–94.

[73] B. Rogers, C. Yan, S. Chhabra, M. Prvulovic, and Y. Solihin, "Single-level integrity and confidentiality protection for distributed shared memory multiprocessors," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 2008, pp. 161–172.

[74] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

[75] A. Rudoff, "Persistent memory programming," *Login: The Usenix Magazine*, vol. 42, pp. 34–40, 2017.

[76] G. Saileshwar, P. Nair, P. Ramrakhyani, W. Elsasser, J. Joao, and M. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 416–427.

[77] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 454–465.

[78] M. I. Seltzer, K. Bostic, M. K. McKusick, C. Staelin *et al.*, "An implementation of a log-structured file system for unix." in *USENIX Winter*, 1993, pp. 307–326.

[79] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent hardware management of stacked dram as part of memory," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 13–24.

[80] S. Swami and K. Mohanram, "Arsenal: Architecture for secure non-volatile memories," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 192–196, 2018.

[81] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 665–678.

[82] A. Tech, "Nvdimm messaging and faq," 2014.

[83] P. Von Behren, "Nvml: implementing persistent memory applications," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST). Santa Clara, California, USA*, 2015.

[84] C. Wang, Q. Wei, J. Yang, C. Chen, Y. Yang, and M. Xue, "Nv-dedup: High-performance inline deduplication for non-volatile memory," *IEEE Transactions on Computers*, vol. 67, no. 5, pp. 658–671, 2017.

[85] R. Wang, Y. Zhang, and J. Yang, "Cooperative path-oram for effective memory bandwidth sharing in server settings," in *High Performance Computer Architecture (HPCA)*, 2017.

[86] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 225–236.

[87] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies FAST 16)*, 2016, pp. 323–338.

[88] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 179–190.

[89] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 331–345.

[90] M. Ye, C. Hughes, and A. Awad, "Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 403–415.

[91] M. Ye, K. Zubair, A. Mohaisen, and A. Awad, "Towards low-cost mechanisms to enable restoration of encrypted non-volatile memories," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[92] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*. IEEE, 2012, pp. 337–344.

[93] V. Young, P. J. Nair, and M. K. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 33–44, 2015.

[94] L. Zhang and S. Swanson, "Pangolin: A fault-tolerant persistent memory programming library," in *2019 USENIX Annual Technical Conference ATC 19)*, 2019, pp. 897–912.

[95] W. Zhang and T. Li, "Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2009, pp. 101–112.

[96] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2013, pp. 421–432.

[97] J. Zhou, A. Awad, and J. Wang, "Lelantus: Fine-granularity copy-on-write operations for secure non-volatile memories," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 597–609.

[98] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 91–104.

[99] K. A. Zubair and A. Awad, "Anubis: ultra-low overhead and recovery time for secure non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 157–168.

[100] P. Zuo and Y. Hua, "Secpm: a secure and persistent memory system for non-volatile memory," in *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[101] P. Zuo, Y. Hua, and Y. Xie, "Supermem: Enabling application-transparent secure persistent memory with low overheads," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 479–492.

[102] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo, "Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes," in *2018 51st Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2018, pp. 442–454.